Improved evolutionary algorithm design for the project scheduling problem based on runtime analysis

Minku, Leandro; Sudholt, Dirk; Yao, Xin

DOI:
10.1109/TSE.2013.52

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (Harvard):
Minku, L, Sudholt, D & Yao, X 2014, 'Improved evolutionary algorithm design for the project scheduling problem based on runtime analysis', *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 83-102. https://doi.org/10.1109/TSE.2013.52

Improved Evolutionary Algorithm Design for the Project Scheduling Problem Based on Runtime Analysis

Leandro L. Minku, Dirk Sudholt, and Xin Yao, *Fellow, IEEE*

Abstract—Several variants of evolutionary algorithms (EAs) have been applied to solve the project scheduling problem (PSP), yet their performance highly depends on design choices for the EA. It is still unclear how and why different EAs perform differently. We present the first runtime analysis for the PSP, gaining insights into the performance of EAs on the PSP in general, and on specific instance classes that are easy or hard. Our theoretical analysis has practical implications—based on it, we derive an improved EA design. This includes normalizing employees' dedication for different tasks to ensure they are not working overtime; a fitness function that requires fewer pre-defined parameters and provides a clear gradient towards feasible solutions; and an improved representation and mutation operator. Both our theoretical and empirical results show that our design is very effective. Combining the use of normalization to a population gave the best results in our experiments, and normalization was a key component for the practical effectiveness of the new design. Not only does our paper offer a new and effective algorithm for the PSP, it also provides a rigorous theoretical analysis to explain the efficiency of the algorithm, especially for increasingly large projects.

Index Terms—Schedule and organizational issues, evolutionary algorithms, software project scheduling, software project management, search-based software engineering, runtime analysis

1 Introduction

Software project scheduling is traditionally one of the major problems faced by software project managers [2]. It is a particularly demanding task [3] and, being a project planning task, is vital to many software engineering activities [4]. Its process involves several duties [3]: identify project activities; identify activity dependencies; estimate resources for activities; allocate people to activities; and create project charts. As explained by Sommerville [3], project scheduling "involves separating the total work involved in a project into separate activities and judging the time required to complete these activities. Usually, some of these activities are carried out in parallel. Project schedulers must coordinate these parallel activities and organize the work so that the workforce is used optimally."

In this work, we concentrate on the problem of optimally allocating people (employees) to activities (tasks) using automated approaches. Similar to previous work [5], [6], [7], we will refer to this problem as the project scheduling problem (PSP). Typical objectives to be optimized when making allocations are to minimize the cost and completion time of the software project. When allocating employees to tasks, employees can divide their attention among several tasks at the same time in the PSP. This is often modelled by dedication values [6] (Chang et al. [5] call it *commitment quotas*) representing the percentage of time an employee spends on a certain task. Employees have a maximum dedication, which must be respected to avoid overwork. The PSP also has other constraints. For instance, the team assigned to a certain task must have all the skills required for this task, and task precedence constraints among tasks must be respected. All these factors contribute to the difficulty of the PSP, making it more complex than classical scheduling problems.

The PSP is particularly challenging when the project is large. The space of possible allocations of employees to tasks is enormous, and providing an optimal allocation of employees to tasks becomes a very difficult task [4]. So, it is desirable to have automated methods to provide near optimal schedules that meet all constraints and can aid a software project manager in his/her decision of what schedule to adopt in order to minimize cost and completion time of the project. In addition, budget and schedule compression has become the norm in today's software industry [8], [9], as a result of its intensified competitiveness due to movements such as globalization, outsourcing and open-source software [10], [11], [12], and [13]. As software companies have to strive to get their jobs done faster for less money in order to succeed [8], automated scheduling tools that can provide insight into how to reduce cost and completion time become increasingly important [14]. The advantages of automated methods are:

- they help the software manager in producing schedules that meet all constraints;

- *L.L. Minku and X. Yao are with CERCIA, School of Computer Science, The University of Birmingham, Birmingham B15 2TT, United Kingdom. E-mail: {L.L.Minku, X.Yao}@cs.bham.ac.uk.*
- *D. Sudholt is with the Department of Computer Science, University of Sheffield, Sheffield S1 4DP, United Kingdom. E-mail: d.sudholt@sheffield.ac.uk.*

Manuscript received 28 Dec. 2012; revised 20 June 2013; accepted 19 Oct. 2013; date of publication 25 Oct. 2013; date of current version 24 Feb. 2014. Recommended for acceptance by T. Menzies.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2013.52

- they provide a very useful insight into how to optimize objectives such as cost and completion time, as they may find solutions that no human has thought of; and
- they speed up the task of allocating employees to tasks, as they can generate schedules taking far less time than a human would if he/she had to plan the whole schedule from the beginning.

With Search-Based Software Engineering as a growing field in Software Engineering [15], evolutionary algorithms (EAs) [16] have been used to solve the PSP. Evolutionary algorithms mimic the natural evolution of species, "evolving" a population of candidate solutions, i.e. project schedules. Different EAs were adopted for different variants of the PSP, e.g., [5], [6], [17], [18], [19]. This also includes multiobjective variants of this problem [4], [7], [20], [21] as well as approaches using cooperative coevolution [22]. A comprehensive survey of related work was presented in the work by Di Penta et al. [4].

In order for search-based software engineering to be applied in the real-world successfully, we need to understand fundamentally what search algorithms can and cannot do. However, despite the existing research, it is still not well understood how design choices in EAs affect performance for the PSP, including what problem features make the PSP a challenging problem for EAs. This lack of understanding makes it hard to make appropriate design choices for EAs to solve the PSP, and it impedes the design of better EAs for this problem. Fundamental understanding of algorithms used in software engineering is essential to advance the science. Some progress was made in the well-known work by Alba and Chicano [6]. They presented an effective EA and a systematic empirical performance analysis of the impact of problem features on the performance of their EA.

Our work presents a novel perspective for designing EAs for the PSP, based on rigorous mathematical arguments. We use runtime analysis to gain insight into how EAs perform on illustrative instance classes. This sheds light on what instances are easy, and which ones are hard to solve for EAs. It also helps to make more informed design choices for the PSP. Based on that, we propose a new design that is shown to be very effective both theoretically and empirically. Our design is thoroughly analyzed and shown to outperform others considering several criteria that may influence a software manager's choice of PSP algorithm. Such an analysis is important to software engineering in practice because applying an algorithm/tool without understanding its benefits/limitations can lead to wrong conclusions.

This paper is organized as follows. Section 2 briefly explains the full process of software project scheduling, and how the duty of allocating employees to tasks (PSP) is integrated into it. Section 3 presents a formal definition of the PSP investigated in this paper. Section 4 explains previous works on using search-based techniques for the PSP. Section 5 presents our search-based approach for solving the PSP. Section 6 presents a runtime analysis to gain insights into the performance of EAs on the PSP in general, and on specific problem instances that are easy or hard. This rigorous theoretical analysis explains the efficiency of our approach. Section 7 provides an experimental analysis to provide the software project manager with insight into what algorithm to chose for the PSP. It gives evidence demonstrating what algorithms are likely to behave better according to different evaluating criteria that may affect the project manager's decision. Section 8 provides an additional discussion on the comparison of different approaches. Section 9 presents the conclusions and future work.

2 The Full Software Project Scheduling Process

As explained in Section 1, the full software project scheduling process involves several duties [3]: identify project tasks; identify task dependencies; estimate resources for tasks; allocate employees to tasks; and create project charts. This paper investigates the duty of allocating employees to tasks, which is called PSP for briefness. This section explains how the PSP is integrated into the full process of software project scheduling.

The PSP uses several pieces of information, which are determined in the prior duties of identifying tasks, identifying task dependencies, and estimating resources for tasks (estimating the effort required for the task, e. g., in person-months). The identification of tasks and their dependencies is performed by the software manager, possibly with the help of the software architect. After that, the software manager provides an estimate of the effort required to complete each of these tasks. Models such as COCOMO 81 [23] can be used as support tools in predicting the required effort based on features such as the estimated software size, development type and several cost drivers, such as required reliability, size of database, etc. Several other methods to estimate software effort exist (e.g., [24], [25], [26], [27], [28]). Additionally, the PSP also uses information about the employees, e.g., which employees are available for the project, and what their salaries and skills are.

After these prior duties are completed, one can proceed to the duty of allocating employees to tasks (PSP). This paper is concerned with the problem of allocating employees to tasks in such a way to minimize the cost and completion time of the software project. In order to use an algorithm to search for such good allocations, a method to estimate the cost and completion time of the project is necessary. Several cost and completion time estimation methods exist [3], [23], [26], [29]. A feature that is common to most of them is the use of the estimated required effort as one of the main ingredients to estimate cost and completion time. In order to guide the search towards good allocations, the method used by the PSP must consider not only the required efforts, but also the allocations of employees to tasks. Considering the allocations is very important for software projects, because its resources are mainly human resources [30]. In this paper, schedule-driven estimation [29] is used for that purpose.

After the allocation of employees to tasks is decided, project charts can be produced. For instance, Gantt charts can be created using information about the tasks and their dependencies, required efforts and allocations.

3 Problem Formulation

This section provides a formal definition of the PSP as an optimization problem, based on existing work [6]. It consists of the following inputs:

- a set of employees $e_1, \ldots, e_n$ with salaries $s_1, \ldots, s_n$, and sets of skills $\text{skill}_1, \ldots, \text{skill}_n$, respectively,
- a set of tasks $t_1, \ldots, t_m$ with required efforts $\text{eff}_1, \ldots, \text{eff}_m$, and sets of required skills $\text{req}_1, \ldots, \text{req}_m$, respectively, and
- a task precedence graph (TPG)—a directed graph with tasks as nodes and task precedence as edges.

The set of tasks and TPG can be referred to as the project to be scheduled. The goal is to assign employees to tasks in order to minimize the completion time as well as the total cost for the project (i. e. salaries paid). Employees can work on several tasks simultaneously, as indicated by their *dedication* to certain tasks. The dedication is the fraction of their time devoted to a particular task. The completion time is the time the project is completed. It is computed with a simple iterative algorithm (described in detail as Algorithm 1 in Section 5.2) that takes into account the task precedence graph, the effort for each task, and the dedication for all employees.

The amount of dedication of an employee e_i to a task t_j is determined by a value $x_{i,j} \in \{0/k, 1/k, \ldots, k/k\}$, where $k \in \mathbb{N}$ reflects the granularity of the solution. For $k = 1$ we only have dedications 0 and 1. For, say, $k = 10$ we have $k + 1 = 11$ dedications of $0, 10, \ldots, 100$ percent. Employees can only work on a task t_j if all employees working together have all the skills to do the task. More formally, we require

$$\text{req}_j \subseteq \bigcup_{i=1}^{n} \{\text{skill}_i \mid x_{i,j} > 0\}. \quad (1)$$

A maximum dedication for each employee was considered previously [6]. It reflects how much of a full-time job an employee is able to work. This can reflect both part-time jobs as well as the willingness to work overtime. Alternatively, one could introduce the *productivity* of an employee as a general indicator of her/his performance. The productivity of a part-time employee could then be decreased accordingly. It would also be feasible to associate different productivities with each skill so as to reflect how good an employee is at using a particular skill. This was done, for instance, in [18]. However, for the sake of simplicity we leave these aspects for future work and consider that all employees have the same maximum dedication of 1.

4 Previous Work on Search-Based Techniques to the PSP

4.1 EAs for the PSP

Search-based techniques such as EAs [16] have been showing success in solving resource-constrained scheduling problems [31]. However, such resource-constrained scheduling problems do not present some features that are important in software projects, e.g., the fact that employees can divide their attention among different tasks at the same time.

The software engineering literature on PSP is more recent than the literature on resource-constraint scheduling problems. For instance, Chang et al. [5] proposed an EA for a problem formulation similar to that in Section 3. One of the main differences is that they considered overwork as an extra objective to be minimized.

Alba and Chicano [6] used the problem formulation explained in Section 2 and proposed an EA which will be referred to as the Genetic Algorithm (GA) in this paper. Under this formulation, employees cannot exceed their maximum total dedication to tasks given by the company's policy, but any amount of overwork within the company's rules is allowed. Later on, Luna et al. [7] investigated a similar problem formulation in a multi-objective scenario where cost and completion time were not combined into a single fitness function. Their work also introduced a repair operator to improve finding feasible solutions. These algorithms are explained in more detail in Sections 4.2 and 4.3.

Di Penta et al. [4] formulated the PSP as the problem of assigning employees to teams and deciding on the order that work packages are selected to be processed in a queue system. The queue system is then used to assign work packages to teams. Different search-based techniques such as single and multi-objective EAs were investigated. The objectives are to minimize completion time and/or fragmentation, rather than completion time and cost. Fragmentation is the total amount of idle person-months in the schedule, due to the precedence constraints of the work packages to be processed.

Chang et al. [19] introduced a time-line to break down a task into smaller components of time-sliced activities. In this way, tasks can be interrupted after they start, and employees do not necessarily need to work on a certain task from its beginning until its end. Several other features were also introduced, such as distinction between an employee and a contractor, different rates of payment depending on whether there is overwork, different proficiencies of the skills, possibility of training during the project, etc. The inclusion of all these features creates a problem formulation closer to reality, but it also introduces a large number of subjective input values required by the algorithm. It is unknown to what extent the solutions provided by the EA employed in their study are sensitive to the uncertainty of these inputs.

Other examples of EAs for the PSP include the work of Kapur et al. [18], multi-objective EAs [20], [21] and cooperative coevolution algorithms [22]. A comprehensive survey of related work was presented in the work by Di Penta et al. [4].

Despite all the existing research, it is still not well understood how design choices in EAs affect performance, including what problem features make the PSP hard for EAs. This lack of understanding impedes the design of better EAs. In our work, we first tackle this issue theoretically considering the problem formulation proposed by Alba and Chicano [6] due to its popularity. A new design is then proposed to overcome problems specific to the existing algorithms designed for this problem formulation. Other problem formulations can be considered as future work.

4.2 GA for the PSP

Alba and Chicano [6] proposed a GA where candidate solutions for the PSP are represented as a matrix of binary values which encode the dedications $x_{i,j}$ for each employee e_i and task t_j. The recombination operator is a 2D single point crossover, which randomly selects a row and a column and then swaps the elements in the upper left and in the lower right quadrant of the two parents. They used bit-flip mutation, binary tournament selection of two parents for recombination, and survival selection based on elitist replacement of the worst individual.

A candidate solution is considered infeasible if there is a task with no employee associated, or the skills constraint (eq. 1) is not satisfied, or there are employees working overtime. Overtime can happen when tasks are executed in parallel, as the total dedication of employees in the candidate solution can exceed 1. Based on that, the fitness function is defined as follows:

$$f(x) = \begin{cases} 1/q & \text{if the solution is feasible,} \\ 1/(q+p) & \text{otherwise,} \end{cases}$$

where $q = w_{\text{cost}} \cdot \text{cost} + w_{\text{time}} \cdot \text{time}$, $p = w_{\text{penal}} + w_{\text{undt}} \cdot \text{undt} + w_{\text{reqsk}} \cdot \text{reqsk} + w_{\text{over}} \cdot \text{over}$, and w_{cost}, w_{time}, w_{penal}, w_{undt}, w_{reqsk} and w_{over} are pre-defined parameters, cost and time are the cost and completion time of the solution, undt is the number of tasks with no employee associated, reqsk is the number of skills still required to perform all project tasks, and over is the total amount of overwork time spent by all employees during the project.

Their fitness function penalizes infeasible solutions, but whether or not it gives hints as to how to reach feasible solutions depends on the chosen values for several predefined parameters (w_{penal}, w_{undt}, w_{reqsk} and w_{over}). So, a wrong parameter choice could make the algorithm unable to guide the search towards feasibility. Moreover, their experimental analysis reveals that the overwork constraint can cause their GA to have very low hit rate in finding feasible solutions, i.e., the algorithm is unable to find feasible solutions in several occasions. This holds even for very simple instances where the skill constraints have been removed (every employee has all skills) and all employees have the same salary.

4.3 A Repair Mechanism

After the preliminary version of this paper [1] was prepared, we learned that Luna et al. [7] independently proposed a *repair mechanism* that facilitates the search for feasible schedules without overwork. Their repair mechanism considers the maximum total dedication of any employee at any point of time during the generated schedule:

$$M := \max_{i,\tau} \left\{ e_i^{\text{work}}(\tau) \right\} + \varepsilon,$$

where $e_i^{\text{work}}(\tau)$ is the total dedication of employee i at time τ, and $\varepsilon = 0.00001$ is a small constant used to prevent floating point inaccuracies. If $M > 1 + \varepsilon$, i. e., if there is overwork, dedications of all employees on all tasks are scaled down by dividing them by M. This implies that overwork is eliminated, but this is done at the expense of increasing the execution time of all tasks by a factor of M.

In many settings this approach might be suboptimal as dedications are unnecessarily reduced during the periods of the schedule where there is no overwork, as long as there is overwork in at least some period of the project.

5 Our Approach

In order to overcome problems related to the hit rate and to improve solution quality, we propose an improved design, which consists of two main points. The first one is to normalize employee's dedications (Section 5.1) in order to address the problem of overwork. It provides an alternative solution to the repair mechanism from [7], trying to reduce dedication values only where necessary, instead of reducing *all* dedication values across the board. The second one is to give a clear gradient for searching towards feasibility by introducing a new type of penalty in the evaluation of cost and completion time (Section 5.2). This improved design can be used with different algorithms. Our experimental analysis (Section 7) uses it with the $(1+1)$ EA, RLS and Pop-EA as explained in Section 5.3.

5.1 Normalizing Dedications

We alleviate the problem of overwork and hence remove a crucial obstacle in the search process of EAs by using the following approach (*normalization*): if at some point of time the total dedication of an employee e_i across all active tasks is $d_i > 1$ then her/his dedication for all tasks at this point of time is divided by d_i. This reflects a very natural way of an employee dividing her/his attention to several tasks. Note that we do not normalize "underwork," i. e., total dedications less than 1. This would otherwise remove the possibility of balancing cost versus completion time.

Normalization allows for much more fine-grained schedules as employees can automatically re-scale their dedications as soon as tasks are finished or new tasks are started. For instance, assume there are two tasks t_1, t_2 suitable for employee e_1 under the problem formulation explained in Section 3. Assume also that the employee works on both tasks at overlapping time intervals and neither normalization nor repair are used. If $x_{1,1} + x_{1,2} > 1$, the employee works overtime whenever she/he works on both t_1 and t_2 at the same time. If $x_{1,1} + x_{1,2} \leq 1$, on the other hand, resources are wasted when the employee works on a single task. So, regardless of the values for $x_{1,1}$ and $x_{1,2}$, there will always be overwork or resources wasted—unless both tasks start and finish at exactly the same time. Note that, depending on $x_{1,1}$ and $x_{1,2}$, there could be even both resources wasted when the employee is working on a single task and overwork when working on both tasks at the same time (whenever $x_{1,1} < 1$, $x_{1,2} < 1$, and $x_{1,1} + x_{1,2} > 1$).

With normalization the semantic of dedication values is changed slightly. For any employee working on two or more tasks, equal dedication values mean that she/he should divide her/his attention equally among all tasks. Dedications 0.5 and 1 mean that she/he should devote twice as much time for the second task as for the first task. A smallest possible example is given in Fig. 1. In this example, the schedule with neither normalization

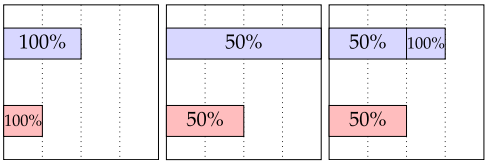


Fig. 1. Three Gantt diagrams, highlighting the dedication of employee e_1 working on two tasks (t_1 and t_2) simultaneously with $x_{1,1} = 1$ and $x_{1,2} = 1$. Left: an infeasible schedule with overwork. Middle: the same schedule after repair, without normalization. Right: the same schedule with normalization.

nor repair is infeasible. In the repaired schedule, the employee uses only 50 percent of her/his time to complete the first task. Normalization, on the other hand, would decrease dedications of 100 percent towards 50 percent as long as two tasks are performed, and then it would automatically remove this scaling factor, so that the dedication of the first task goes back up to 100 percent after the other task is completed. We will come back to the comparison of normalization and repair in Section 8.

5.2 Evaluating Costs and Completion Time

Algorithm 1 makes a schedule-driven estimation of cost and completion time by implicitly constructing a Gantt diagram as explained in the following. If the dedication matrix is the *genotype* during optimization, the final schedule can be called *phenotype*. The phenotype for any feasible genotype is constructed iteratively (lines 8-28), and is assessed by storing the $d_{i,j}$-values from each iteration, along with the corresponding time stamps. The algorithm checks which tasks can be active at the current point of time (line 9). The normalized dedication for all suitable employees is computed for all these tasks (line 15). Then we determine the earliest point of time t at which a task is finished (line 19). All finished tasks are being marked as finished (line 25), so that the next iteration can include new tasks, according to the task precedence graph. If there are tasks that have been started, but are not finished yet, their effort is updated to the remaining effort needed for completing them (line 23). This accounts for potentially piecewise evaluations of particular tasks. Along the way, all completion times and costs in all iterations are added up (lines 20-21). The total completion time and cost are the output when all tasks have been processed (line 29).

Before constructing the schedule, Algorithm 1 tests whether the genotype is infeasible in that not every task can be completed in finite time (lines 1-4). That is the case when the skills constraint (eq. 1) is not satisfied. Note that this includes the case in which there is a task with no employee designated, as tasks with no employee associated can be seen as tasks lacking all the required skills. If the genotype is infeasible, the algorithm returns very high costs and completion times (line 6): $(\mathrm{reqsk} \cdot 2\sum_{i=1}^{n}\sum_{j=1}^{m} s_i \mathrm{eff}_j, \mathrm{reqsk} \cdot 2k\sum_{j=1}^{m}\mathrm{eff}_j)$, where reqsk is the number of required skills missing. These values are chosen in such a way that any improvement in decreasing the number of missing skills results in a solution that has both lower costs and lower completion times:

Lemma 1. *The output of Algorithm 1 is such that for every two solutions x, y, if y has fewer missing skills than x (including no missing skills) then* $\mathrm{cost}(y) < \mathrm{cost}(x)$ *and* $\mathrm{time}(y) < \mathrm{time}(x)$.

Algorithm 1 Evaluate$(\mathrm{cost}, \mathrm{time}, \mathrm{TPG})$
Output: $(\mathrm{cost}, \mathrm{time})$

1: Let $\mathrm{reqsk} = 0$.
2: **for** all tasks t_j **do**
3: $\quad \mathrm{reqsk} = \mathrm{reqsk} + |\mathrm{req}_j \setminus \bigcup_{i=1}^{n}\{\mathrm{skill}_i \mid x_{i,j} > 0\}|$.
4: **end for**
5: **if** $\mathrm{reqsk} > 0$ **then**
6: $\quad$ Output $\left(\mathrm{reqsk} \cdot 2\sum_{i=1}^{n}\sum_{j=1}^{m} s_i \mathrm{eff}_j, \mathrm{reqsk} \cdot 2k\sum_{j=1}^{m}\mathrm{eff}_j\right)$; stop.
7: **end if**
8: **while** $\mathrm{TPG} \neq \emptyset$ **do**
9: $\quad$ Let V' be the set of all unfinished tasks without incoming edges in TPG.
10: $\quad$ **if** $V' = \emptyset$ **then**
11: $\quad\quad$ Output "Problem instance not solvable!" and stop.
12: $\quad$ **end if**
13: $\quad$ **for** all tasks t_j in V' **do**
14: $\quad\quad$ **for** all employees e_i **do**
15: $\quad\quad\quad$ Let $d_{i,j} := \frac{x_{i,j}}{\max\left(1, \sum_{t_\ell \in V'} x_{i,\ell}\right)}$.
16: $\quad\quad$ **end for**
17: $\quad\quad$ Compute the total dedication $d_j := \sum_{i=1}^{n} d_{i,j}$.
18: $\quad$ **end for**
19: $\quad$ Let $t := \min_j(\mathrm{eff}_j / d_j)$.
20: $\quad$ Let $\mathrm{cost} := \mathrm{cost} + t\sum_{i=1}^{n} s_i \sum_{j=1}^{m} d_{i,j}$.
21: $\quad$ Let $\mathrm{time} := \mathrm{time} + t$.
22: $\quad$ **for** all tasks t_j in V' **do**
23: $\quad\quad$ Let $\mathrm{eff}_j := \mathrm{eff}_j - t \cdot d_j$.
24: $\quad\quad$ **if** $\mathrm{eff}_j = 0$ **then**
25: $\quad\quad\quad$ Mark t_j as finished and remove it and its incident edges from TPG.
26: $\quad\quad$ **end if**
27: $\quad$ **end for**
28: **end while**
29: Output $(\mathrm{cost}, \mathrm{time})$ and stop.

A proof of this lemma is given in the appendix. This implies that any decrease in the number of required skills missing strictly decreases both components through reqsk. When cost and completion times are used in any reasonable single- or multi-objective fitness function, this gives a clear gradient for search algorithms towards feasibility, without the need for pre-defined parameters to guide the search towards feasibility as done in previous work [6]. It is also worth noting that this strategy deals both with solutions that are infeasible due to the skills constraint and due to tasks with no employee associated. An EA using it can have a very simple fitness function which does not need to treat these two types of infeasible solution separately as done in previous work [6].

5.3 New EA Designs

When designing an EA, one needs to chose the encoding (internal representation of candidate solutions), mutation (operator to create a new candidate solution by modifying an existing one), crossover (optional operator to create new candidate solutions by combining existing ones), and fitness function (procedure to evaluate the quality of a candidate solution). This section presents our design, as well as the

algorithms used in our study. For general knowledge on EAs, we refer the reader to [16].

5.3.1 Encoding and Mutation

A binary encoding was used previously [6] to represent $x_{i,j}$. This however means that the probability of turning one numerical value into another by mutation depends on the binary encoding of these two values. As a result, some mutations (e. g., mutating $7/k \triangleq 0111_2$ towards $8/k \triangleq 1000_2$ flipping 4 bits) are less likely than others (e. g., mutating $8/k \triangleq 1000_2$ towards $9/k \triangleq 1001_2$ flipping 1 bit).

We follow a different approach and work with a direct encoding of dedication values: the search space is $\{0/k, \ldots, k/k\}^{nm}$. Mutation is performed by changing each entry $x_{i,j}$ of the dedication matrix with probability $P_m = 1/(nm)$, independently from other entries. Changing an entry $x_{i,j}$ means replacing it by a value chosen uniformly at random from $\{0/k, 1/k, \ldots, k/k\} \setminus \{x_{i,j}\}$. This implies that each new value has the same chance of being created by mutation. Note that mutation has a chance of changing several dedication values at the same time, changing only a single entry, or not changing any entries at all.

5.3.2 Crossover

In this paper, the crossover operator uses two parent candidate solutions to create two offspring candidate solutions by selecting one of the following two strategies with an equal probability:

- For each employee, select its corresponding dedications to tasks from one randomly chosen parent to compose the first child, and from the other parent to compose the second child. This can be seen as exchanging "rows" of employee's dedications if we visualize the values $x_{i,j}$ as a matrix of employees by tasks.
- For each task, select its corresponding employees' dedications from one randomly chosen parent to compose the first child, and from the other parent to compose the second child. This can be seen as exchanging "columns" of dedications to tasks.

The above is different from the crossover operator used before [6]. The reason is that their operator, which is a 2-D single point crossover, allows part of the dedications of an employee to tasks to be inherited by one child and the remaining part by the other child. So, if a solution is good because the dedications of a certain employee are well balanced for the problem instance, crossover can break this balance by mixing these dedications with dedications of another employee. An illustrative example of that in their algorithm is the fact that, even if the parents do not contain overwork, crossover could easily generate children with overwork. A similar problem could happen to the dedications of all employees to a certain task. Our crossover operator either exchanges full rows of employee's dedications or full columns of dedications to tasks, in an attempt to avoid breaking balances that may be important for the solutions. As it is not known a priori whether the balance in terms of rows or columns is more important, our crossover gives 50 percent of chance for exchanging rows and 50 percent for columns.

5.3.3 Fitness Function

A very simple fitness function can be used based on Algorithm 1. The fitness function (to be minimized) used in our experimental analysis is $f(x) = w_{\text{cost}} \cdot \text{cost} + w_{\text{time}} \cdot \text{time}$, where cost and time are obtained from Algorithm 1. The only pre-defined parameters in this fitness function are the weights w_{cost} and w_{time}, which are used to adjust the desired relative importance of cost and completion time. So, it is easier to configure than other fitness functions [6]. The runtime analysis is not restricted to this fitness function.

5.3.4 Algorithms

As our main goal is to introduce runtime analysis for the PSP, we follow many previous examples in this area [32], [33], [34], [35] and use one of the simplest EAs in our theoretical studies: the $(1+1)$ EA. It is simple because it uses a "population" of just one individual and it does not use crossover. One generation consists of mutation, and then selection checks whether the mutation has found an improvement. Algorithm 2 describes the $(1+1)$ EA for our search space, for minimizing some fitness function f. The encoding, mutation and fitness function used in this paper are described in Sections 5.3.1 to 5.3.3.

Algorithm 2 (1+1) EA for project scheduling

```
1: Initialise x.
2: repeat
3:    Create x' by copying x.
4:    Apply mutation to x' using probability P_m.
5:    if f(x') ≤ f(x) then x := x'.
6: until happy
```

In practice, like all EAs, it will be stopped either after a certain amount of time has elapsed, or when a satisfactory solution has been found. For a theoretical analysis it is a common practice to consider an EA as an infinite process and to measure the expected time needed to find a global optimum. Note that the expected number of components changed in one iteration equals 1. The $(1+1)$ EA is a bare-bone version of EAs and captures their core working principles in a very clear fashion. It can also be a very effective EA, when considering the number of fitness evaluations as the performance measure. For well-known functions like OneMax and LeadingOnes the $(1+1)$ EA is the best EA that only uses mutation for variation [36].

We will also investigate *randomized local search (RLS)*. It differs from the $(1+1)$ EA in that during mutation exactly one dedication value is changed. The entry is chosen uniformly at random. So, RLS can only make local steps, i. e., it can easily get stuck in a local optimum. The $(1+1)$ EA has a global search operator, which allows larger jumps to be made, so that the $(1+1)$ EA can, in principle, escape from any local optimum. However, large jumps are only made with a small probability.

From a theoretical perspective the $(1+1)$ EA is harder to analyze than RLS. The reason is that a mutation changing multiple dedication values simultaneously may have both beneficial and detrimental effects. The mutant will be accepted in the light of detrimental mutations if the combined effect of all changes on fitness is non-negative. There are situations where a mutation is accepted, even though

TABLE 1
Summary of the Algorithms Used in Our Study

Algorithm	Encoding	Mutation	Crossover	Parent selection	Survival selection	Population size	Normalisation
(1+1) EA	Direct	Uniform	–	–	Elitist	1	Yes
RLS	Direct	Uniform 1 entry	–	–	Elitist	1	Yes
Pop-EA	Direct	Uniform	Rows/columns	λ binary tournaments[1]	Elitist	μ	Yes
(1+1) EA no-norm	Direct	Uniform	–	–	Elitist	1	No
GA [6]	Binary	Bit-flip	2-D single point	2 binary tournaments[1]	Elitist	μ	No

[1] *Number of tournaments per generation.*

the mutant is "further away" from the optimum, in a sense that less dedication values coincide with the respective values in the global optimum.

In order to check whether improvements in solution quality can be obtained by using a population, our experimental analysis also considers the $(\mu + \lambda)$ population-based EA (Pop-EA) presented in Algorithm 3. It differs from the $(1 + 1)$ EA in that it maintains a population of μ candidate solutions, rather than a single solution. At each generation, λ parents are selected based on binary tournaments. Crossover is applied to each pair of parents with a pre-defined probability P_c. Mutation is then applied to each child as in the $(1 + 1)$ EA. The mutation and crossover operators used in this paper are explained in Sections 5.3.1 and 5.3.2, respectively. An elitist approach that retains the best μ candidate solutions among the parents and children is used to select candidate solutions that survive for the next generation.

Algorithm 3 Pop-EA for project scheduling

1: Initialise population P with μ candidate solutions.
2: **repeat**
3: Select λ parents from P using binary tournament selection.
4: **for** each pair of parents $x^{(1)}$ and $x^{(2)}$ **do**
5: With probability P_c, apply crossover between $x^{(1)}$ and $x^{(2)}$ to generate $x'^{(1)}$ and $x'^{(2)}$.
6: Otherwise, $x'^{(1)} \leftarrow x^{(1)}$ and $x'^{(2)} \leftarrow x^{(2)}$
7: Apply mutation to $x'^{(1)}$ and $x'^{(2)}$ using probability P_m.
8: $P \leftarrow P \cup \{x'^{(1)}, x'^{(2)}\}$.
9: **end for**
10: Select the μ best candidate solutions from P to survive for the next generation, based on the fitness function f.
11: **until** happy

A summary of these algorithms is presented in Table 1. This table also shows two other algorithms that will be used in Section 7.

6 Runtime Analysis

In the following, we estimate theoretically the *optimization time* of the $(1 + 1)$ EA and RLS, defined as the first generation in which a global optimum is found.

The only assumption we make about the fitness function f is that it is *Pareto-compliant* in a strict sense: if x' Pareto-dominates x (i.e., $\text{cost}(x') \leq \text{cost}(x) \wedge \text{time}(x') \leq \text{time}(x)$) and x does not Pareto-dominate x' then $f(x') < f(x)$. That is, any improvement in one or both objectives also improves the fitness f. This is the case, for instance, when the fitness is chosen as any weighted combination of cost and completion time, and our results are independent of the choice of these weights. In the special case where all employees have the same salary, the costs are always the same. Then f boils down to minimizing the completion time; and as $(1 + 1)$ EA and RLS only depend on the order of search points, and not on their absolute fitness values, we can then, without loss of generality, take $f(x) = \text{time}(x)$.

For details about asymptotic notation (O, Ω, Θ) used in the following, we refer the reader to [37].

6.1 Optimal Completion Times

The two goals of minimizing costs and completion time are often conflicting. We first look at the extreme case of minimizing the completion time only. In every feasible solution for each task the team has the required skills for the task. This also holds if more employees join in working on a task, regardless of their skills. The following theorem describes the optimal solutions to this special case.

Theorem 2. *For every solvable PSP instance, the completion time is minimal if in the schedule all employees always work full time. Then the completion time is* $1/n \cdot \sum_{j=1}^{m} \text{eff}_j$.

If normalization is used, a sufficient condition for minimality is that all dedication values are 1.

The proof of the first statement is straightforward as all employees will work full time until the project is completed. It is obvious that this minimizes the completion time.

The second statement is not true without normalization, as the all-1s matrix will lead to massive overwork on almost all instances. Without normalization, the difficulty for optimization is to find the ideal balance between different tasks, while avoiding overwork. When normalization is used, this difficulty disappears.

6.2 General Time Bounds

This section presents general time bounds for arbitrary instances of the PSP and almost arbitrary EAs. The considerations herein are based on the progress EAs, and, specifically, local and global mutation operators, can make at the genotype level.

We first present a lower bound on the expected running time, in case the problem only allows for a single optimal solution. This lower bound indicates the least time we should allow for running an EA on the problem. In a more general sense, the analysis applies to the time for finding any fixed target point.

Theorem 3. *Consider a PSP instance with n employees and m tasks, $nm > 1$, with a single global optimum (a fixed target) in the fitness function used. Consider an EA starting with a population of random solutions (of arbitrary size) and afterward creating new solutions from previously created ones by local or global mutations. The expected optimization time (time to hit the target) of the EA, with or without normalization, is at least $\Omega(knm\log(nm))$.*

A formal proof is given in the appendix. In simple terms, the analysis shows that initially many dedication entries will differ from their optimal values. In order to find the global optimum (the target point), the EA needs sufficient time to perform a successful mutation on all these entries, where successful means that the correct value is being chosen. This takes the time claimed in the theorem.

Theorem 3 does not guarantee that an EA will find an optimum in the stated time, it just says that every EA needs *at least* the stated time, and maybe much longer. Depending on the problem instance, the EA may get stuck in local optima, particularly if only local mutations are being used.

In contrast, global mutations guarantee that any particular solution can be created from any other one. This means that an EA using global mutations will eventually find a global optimum, albeit this time can be exponential. A proof of the following theorem is given in the appendix.

Theorem 4. *Every EA that constructs new solutions using global mutations (potentially after applying recombination and/or other operators) finds a global optimum for the PSP, with or without normalization, in an expected number $(knm)^{nm}$ of constructed solutions.*

Note that this upper time bound is very crude—it is larger than the expected optimization time of random search or exhaustive search. Many instances can be solved much faster, as we will see in Section 6.4. However, Theorem 4 reflects the truth that EAs may indeed perform worse than random search. This happens when the algorithm gets trapped in a local optimum which is very dissimilar to all global optima, hence requiring many dedication values to be changed simultaneously in one mutation. This is hard to achieve for EAs. Section 6.5 contains an example of a hard local optimum, where exponential time is needed to escape from it (albeit the time is smaller than the bound from Theorem 4).

6.3 Time to Feasibility

In order to see how efficiently our treatment of infeasible solutions guides evolution towards feasible search points, we estimate the expected time until feasibility is reached. The following theorem makes a very reasonable assumption: the total number of skills is not larger than some polynomial in nm.

Theorem 5 ([1]). *Consider RLS or the $(1+1)$ EA, with or without normalization, on any solvable instance where $\sum_{i=1}^{n}|\text{skill}_i| \leq (nm)^{\delta}$ for some constant $\delta \geq 0$. The expected time until some feasible schedule is found is $O(nm\log(nm))$.*

For a proof we refer to [1]. The upper bound is by a factor of k smaller than the lower bound from Theorem 3. This indicates that the time to feasibility is only a small fraction of the optimization time.

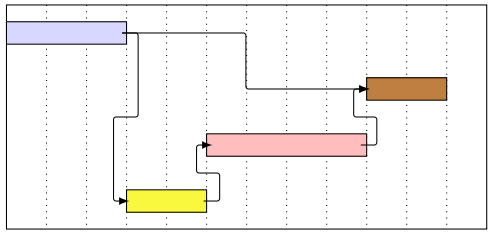


Fig. 2. Example of a linear schedule: all tasks have to be processed sequentially.

6.4 Easy Instance Classes: Linear Schedules

Now we turn to a class of illustrative "easy" instances. Assessing how effective an EA is on easy instances is an excellent starting point for an analysis. It gives a good baseline for comparisons with other results on the running time, and most importantly we learn about the search behavior of an EA. We need a good understanding of easy instances before we can move on to more complicated instance classes.

As easy cases we consider schedules with a "linear" structure: the task precedence graph is a chain of m vertices, or more generally any directed acyclic graph that contains such a chain. In this case all tasks have to be completed sequentially. Fig. 2 gives an example. We also assume that all salaries are equal. So the problem boils down to minimizing the completion time.

Note that if normalization is switched on, it is never actually applied as at each time only one task is processed. The issue of whether normalization is used or not is irrelevant for linear schedules. The same holds for the repair mechanism [7].

Also note that linear schedules can be optimized by minimizing the completion times of all tasks individually. This property is called *separability*: a function is *separable* if it can be written as the sum of functions defined on disjoint sets of variables. This is the case for the completion time of (feasible) linear schedules as the completion time can be written as the sum of completion times for all single tasks.

We know from Theorem 2 that the optimal solution is a dedication matrix where all entries are set to 1. This is the only global optimum as otherwise increasing any dedication value $d_{ij} < 1$ would decrease the execution time for the jth task, while keeping the other execution times unchanged.

RLS only changes one dedication entry at a time, which implies that all tasks will be optimized separately. This leads to the following result, which was proven in [1].

Theorem 6 ([1]). *Consider an instance with n employees with equal salaries and m tasks with arbitrary positive efforts. Let the task precedence graph contain an m-vertex chain as subgraph. Then the expected optimization time of RLS, with or without normalization or repair, is of order $\Theta(knm\ln(nm))$.*

We believe that the $(1+1)$ EA is asymptotically as efficient as RLS on linear schedules, i. e., the bounds from Theorem 6 also hold for the $(1+1)$ EA. However, proving this is more difficult than for RLS. The reason is that global mutations can change several dedications at the same time. Mutations with beneficial and detrimental effects can be accepted if there is a net improvement in fitness. Such situations can happen because jobs with large task efforts have a

larger impact on the fitness than jobs with smaller efforts. This can lead to situations where the fitness improves, but the $(1+1)$ EA actually moves away from the global optimum. We are looking for a formal proof that the $(1+1)$ EA is efficient for arbitrary task efforts.

We first prove a polynomial running time bound for the $(1+1)$ EA on any linear schedule. We do not believe that this bound is asymptotically optimal, but it gives a polynomial upper bound for all instances. We then give better upper bounds for special cases, supporting our belief that the $(1+1)$ EA is as efficient as RLS.

Theorem 7. *Consider an instance with n employees with equal salaries and m tasks with arbitrary positive efforts. Let the task precedence graph contain an m-vertex chain as a subgraph. Then the expected optimization time of the $(1+1)$ EA, with or without normalization or repair, is at most $O((knm)^2)$.*

A proof is given in the appendix; it uses a structural result about separable functions [38], saying that under certain conditions the expected time for the $(1+1)$ EA optimizing a separable function is bounded from above by the sum of expected times for minimizing the completion times of all tasks sequentially.

If $k=1$, i.e., only dedications 0 and 1 are allowed, the analysis becomes easier. It is not hard to see that then the fitness function is *monotonic* in the following sense: flipping only 0-bits to 1 and not flipping any 1-bit to 0 results in a strict fitness improvement. Such an operation implies that the total dedication on every task is non-decreasing, and at least one total dedication is increased strictly. By Doerr et al. [39] this yields the following.

Theorem 8. *Consider an instance with n employees with equal salaries and m tasks with arbitrary positive efforts. Let the task precedence graph contain an m-vertex chain as a subgraph. If $k=1$ then the expected optimization time of the $(1+1)$ EA is $O((nm)^{3/2})$. Additionally, if the mutation probability is changed to $c/(nm)$ for a constant $0<c<1$, the expected time is even bounded by $O(nm\log(nm))$.*

With $k=1$ the upper bound of $O((nm)^{3/2})$ is the same as $O((knm)^{3/2})$. Compared to the bound $O((knm)^2)$ from Theorem 7, we have reduced the exponent of the running time. The second statement about reduced mutation probabilities indicates that the upper bound of $O(knm\log(nm))$ for RLS might also hold for the $(1+1)$ EA, as we have no reason to believe that a tiny reduction of the mutation rate by an arbitrary constant factor (e. g., by 0.99) has a dramatic effect on the expected optimization time here.

We consider one more setting which supports our belief. The following result states that whenever we have a fixed number of employees, a special case of dedication values 0 or 1, and an arbitrary number of tasks, then the expected optimization time of the $(1+1)$ EA is upper bounded just like it is for RLS (cf. Theorem 6). The proof uses another structural result about separable functions [38], stating that in some settings the $(1+1)$ EA will optimize all subfunctions of a separable function in parallel.

Theorem 9. *In the setting of Theorem 6, if additionally $k=1$ (i. e., we have dedications 0 or 1) and $n=O(1)$, then the expected optimization time of the $(1+1)$ EA, with or without normalization, is bounded by $O(knm\log(nm))$.*

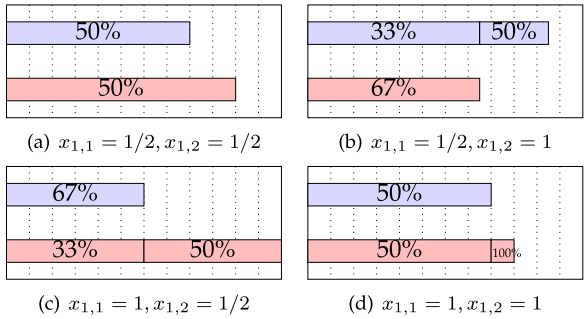


Fig. 3. Gantt diagrams of all feasible schedules for the example from Theorem 10 and the employee's dedication. (a) is a local optimum, (d) is the only global optimum.

The upper bound is even $O(m\log m)$ due to our assumptions on k and n, but to make a fair comparison we write it so that it matches the upper bound for RLS from Theorem 6. Again, the proof is given in the appendix.

All results above support our conjecture that the $(1+1)$ EA is as effective as RLS for linear schedules.

6.5 Difficult Instance Classes

We now look at difficult instances to get insight into what makes the PSP hard. Linear schedules are easy to solve as all tasks are processed sequentially. So we consider settings with tasks being processed in parallel. We have already seen in Section 5.1 that without normalization an EA struggles in finding an optimal balance between dedications for different tasks. With normalization this problem becomes a lot easier. But we show in the following that even with normalization, RLS and the $(1+1)$ EA can still struggle in finding an optimal balance.

In particular, RLS can get stuck in local optima even on a very simple and tiny problem instance where costs are irrelevant. The instance contains two tasks with efforts 4 and 5, respectively. There is only one employee, so trivially we always get a fixed cost for any feasible schedule. Starting with equal dedication values of $1/2$, both tasks finish at similar times. The task with higher effort takes two more time steps, throughout which the employee only works half time, see Fig. 3a. This increases the completion time, compared to the optimal schedule where both dedications are 1, see Fig. 3d. Any local operation either creates an infeasible schedule or increases the imbalance between the two tasks. This increases the time period at the end where the employee only works half time and it makes the schedule even worse, see Figs. 3b and 3c.

RLS has a positive probability of starting in the local optimum, and no local mutation is accepted. Thus, we get:

Theorem 10. *There is an instance with $k=2$, only $m=2$ tasks and just $n=1$ employee (see Fig. 3) where RLS with normalization has an infinite expected optimization time.*

This example shows that the global mutation operator used in the $(1+1)$ EA is important in general, as it may be necessary to change several dedication values at the same time.

Also note that in the instance from Fig. 3 without normalization—with or without repair—an optimal completion

time of $4 + 5 = 9$ can only be achieved if the granularity is set such that dedication values $4/9$ and $5/9$ are feasible choices. If this is not the case (i.e., if k is not a multiple of 9), only completion times larger than 9 are possible. It is easy to see that for every given value of k we can find a similar instance where every feasible schedule without normalization has a strictly larger completion time than the optimal schedule with normalization.

The above instance can also be generalized towards more than two tasks. We do this in such a way that the $(1+1)$ EA has to make a very large mutation in order to escape from a local optimum. Set $\text{eff}_1 = \text{eff}_2 = \cdots = \text{eff}_{m-1} = 3m$ and $\text{eff}_m = 3m + 1$. There are no precedence constraints. Set $k = m$ and $n = 1$, that is, there is only one employee.

Similar to the instance from Theorem 10, setting all dedication values to $1/k = 1/m$ yields a local optimum. All other solutions are worse—except for solutions where all dedications are larger than $1/m$. In order to escape from the local optimum, all dedication values need to be changed in a single mutation. The expected running time then increases exponentially in the number of tasks.

Theorem 11. *For every $m \in \mathbb{N}$ there is an instance with m tasks and one employee and an initial search point for the $(1+1)$ EA with normalization such that its expected optimization time from there is at least m^m.*

A proof is given in the Appendix.

Note that Theorem 11 does not make a claim about the expected optimization time with uniform initialization as the probability of getting stuck in the local optimum stated in the proof might be exponentially small. However, Theorem 11 proves the existence of local optima which are very hard to overcome.

Even though normalization makes it easier to balance dedications, there is still a risk of non-optimal equilibria between dedications for tasks processed in parallel. This can present a major obstacle for EAs as many dedications might need to be changed in a single mutation.

Finally, note that the instance from Theorem 11 can easily be modified by adding further tasks that have to be processed sequentially, and after all existing tasks. Adding these new tasks does change the number of tasks, but it does not significantly affect the expected optimization time. This means that for every given value of m we can mix characteristics of our "hard" and "easy" instances. By deciding how many tasks shall belong to the easy or hard part, we can create instances of tunable difficulty. This shows that there are not only easy and hard instances, but there is a fine scale of expected optimization times that can be observed for EAs.

7 Experimental Analysis

Faced with several different possible choices of algorithms that can be used for the PSP, it is desirable to provide the software project manager with insight into what algorithm to choose. This insight should be followed by evidence demonstrating what algorithms are likely to behave better according to different evaluation criteria that may affect the project manager's decision. With this as the main objective, this section presents an empirical analysis of different algorithms based on the hit rate (number of runs in which a feasible solution was found), solution quality and convergence time. As a secondary objective, this section also provides information that can be used by other researchers trying to improve these algorithms.

The algorithms included in the analysis are our three algorithms with normalization ($(1+1)$ EA, RLS and Pop-EA), and two algorithms without normalization ($(1+1)$ EA no-norm and a state-of-the-art GA [6]),[1] as shown in Table 1. Preliminary results using $(1+1)$ EA have been presented in [1]. We will refer to the algorithms with normalization simply as $(1+1)$ EA, RLS and Pop-EA. When referring to the $(1+1)$ EA without normalization, we will explicitly state that or use the term "no-norm."

The $(1+1)$ EA without normalization works similarly to our $(1+1)$ EA, but with an extra condition on the fitness function to consider overwork. In this case, the fitness is $f(x) = w_{\text{pess}} + \text{over}$ if the skills constraint (eq. 1) is satisfied but there is overwork. The value over is the total amount of overwork time spent by all employees during the project [6] and $w_{\text{pess}} = w_{\text{cost}} \cdot 2\sum_{i=1}^{n}\sum_{j=1}^{m} s_i \text{eff}_j + w_{\text{time}} \cdot 2k\sum_{j=1}^{m}\text{eff}_j$.

This section is further divided as follows: Section 7.1 presents the data sets used in the experiments; Section 7.2 presents the parameters; Section 7.3 presents the results in terms of hit rate; Section 7.4 in terms of solution quality; and Section 7.5 in terms of convergence time.

7.1 Data Sets

In order to make a fair comparison against the GA, we used the same 48 instances (benchmarks 1-5) of the PSP as before [6]. Benchmarks 4-5 can be found at http://tracer.lcc.uma.es/problems/psp/generator.html. Benchmarks 1-3 were made available to us by Enrique Alba. As the number of problem instances is high, we avoid biasing the conclusions and remove the possibility of hand-tuning the algorithm to a particular problem instance. Moreover, we can verify how the algorithms are affected by problem features such as the number of employees, number of tasks, number of employee's skills and number of project demanded skills.

Benchmarks 1-3 were used to analyze the effect of varying each of the three problem features (number of employees, number of tasks, and number of employee's skills) while maintaining the other features fixed. These data sets use the same salary for all employees ($10,000), so that, given a project, the cost of all solutions for this project is always the same. In this way, the ideal cost per unit of time is known and it is possible to evaluate how close a given solution is to the optimum in terms of completion time.

Benchmark 1 is composed of four instances varying the number of employees among 5, 10, 15 and 20. Benchmark 2 is composed of three instances varying the number of tasks among 10, 20 and 30. Benchmark 3 is composed of five instances varying the number of employee's skills among 2, 4, 6, 8 and 10 skills, which are randomly selected from a set of 10 project skills. Each task requires five different skills in this benchmark. In benchmarks 1 and 2, all

1. Our implementation of the $(1+1)$ EA, RLS and Pop-EA were based on the Opt4J framework [40].

TABLE 2
Hit Rate Out of 100 Runs for the $(1+1)$ EA without Normalization, and an Existing GA (Obtained from [6])

(a) Benchmarks 1–3

Benchmark 1			Benchmark 2			Benchmark 3		
Employees	(1+1) EA no-norm	GA	Tasks	(1+1) EA no-norm	GA	Employees' skills	(1+1) EA no-norm	GA
5	97	87	10	97	73	2	0	39
10	100	65	20	84	33	4	0	53
15	97	49	30	3	0	6	24	77
20	96	51	–	–	–	8	11	66
–	–	–	–	–	–	10	100	75

(b) Benchmarks 4–5

Tasks	Benchmark 4				Benchmark 5			
	4-5 employees' skills 5,10,15 employees		6-7 employees' skills 5,10,15 employees		5 project skills 5,10,15 employees		10 project skills 5,10,15 employees	
	(1+1) EA no-norm	GA	(1+1) EA no-norm	GA	(1+1) EA no-norm	GA	(1+1) EA no-norm	GA
10	2,0,89	94,97,97	9,100,100	84,100,97	10,49,90	98,99,100	0,0,0	61,85,85
20	0,2,17	0,6,43	0,78,11	0,76,0	0,2,67	6,9,12	0,0,0	8,1,6
30	0,0,0	0,0,0	0,6,0	0,0,0	0,0,1	0,0,0	0,0,0	0,0,0

The hit rate for the algorithms with normalization was always 100.

employees have all necessary skills, i. e., the skills constraint (eq. 1) is always satisfied. Instances within each of the benchmarks 1 and 3 represent the same project to be developed (i. e., they have the same tasks and TPG) with the number of tasks fixed to 10. Each instance of benchmark 2 represents a different project, as the number of tasks is different. In benchmarks 2 and 3, the number of employees is fixed to 5.

Benchmarks 4 and 5 are composed of instances that represent different projects and each employee has a different salary. Each benchmark is composed of 18 instances which vary all the previous problem features. The number of employees can be 5, 10 or 15 and the number of tasks can be 10, 20 or 30. In benchmark 4, the total number of project skills is 10, and two ranges were considered separately for the number of employees' skills: 4-5 and 6-7. In benchmark 5 the number of project demanded skills can be 5 or 10, and the number of skills per task and employee is in the range 2-3.

7.2 Experimental Setup

For a fair comparison between our algorithms and the GA, we have used the following parameters, which correspond to the parameters used previously in the literature [6]:

- Parameters for all our algorithms:
  - constant for the granularity of the solution $k = 7$;
  - $w_{\text{cost}} = 10^{-6}$;
  - $w_{\text{time}} = 10^{-1}$;
  - number of independent runs per problem instance 100.
- Parameters specific to the non-population-based algorithms:
  - number of generations $5,064$ (= number of fitness evaluations considering the initial population of size 64 used previously [6]).
- Parameters specific to the Pop-EA:
  - size of the population $\mu = 64$, as in [6];
  - number of children generated at each generation $\lambda = 64$;
  - number of generations 79 (this is the rounded number of fitness evaluations in our experiments with the non-population-based algorithms divided by the population size ($79.125 \approx 79$), giving a total of 5,056 fitness evaluations).

The GA [6] always applies crossover, whereas our Pop-EA uses a probability of crossover. This probability was set to 0.75, which is the middle of the crossover rate interval $[0.6, 0.9]$ suggested by Eiben and Smith [16]. The mutation operator provided by the framework Opt4J selects any value in $\{0/k, \ldots, k/k\}$ uniformly at random. This does not exclude the original value, hence each dedication value is only changed with probability $1/(nm) \cdot k/(k+1)$ in our experiments with $(1+1)$ EA and Pop-EA.

7.3 Hit Rate

An algorithm designed for the PSP should ideally provide feasible allocations of employees to tasks. Algorithms unable to provide feasible solutions are not useful for the software project manager. With that in mind, this section analyzes the hit rate (Table 2).

Our three algorithms with normalization always achieved hit rate 100, i. e., all runs always found a feasible solution. The algorithms without normalization frequently presented much lower hit rates, sometimes even a hit rate of zero, i. e., even after running the algorithms 100 times, no feasible solution was found. In order to produce a boundary to capture the unknown population[2] of hit rates, we calculated the modified (adjusted) Wald binomial confidence interval [41] with 95 percent of confidence for each hit rate. For hit rate of 100, the interval is $[96.83, 100.00]$. The upper limits of the confidence intervals for the hit rates of 90 or less shown in Table 2 are all lower than the lower limit of the interval for hit rates of 100, meaning that the differences between hit rates of 90 or less and the hit rate of the algorithms with normalization are statistically significant.

As the $(1+1)$ EA without normalization presented in general much lower hit rates than the $(1+1)$ EA with

2. The term *population* in this sentence refers to the statistical meaning of population, against the known *sample* of hit rates.

TABLE 3
Average Ranking of Algorithms According to Fitness across Problem Instances

∼ Avg Rank	Avg Rank	Std Deviation	Algorithm
1	1.1875	0.5708	Pop-EA
2	2.0833	0.3472	(1+1) EA
3	2.7292	0.6098	RLS
4	4.0000	0.0000	(1+1) EA no-norm

Smaller ranking values are better rankings.

TABLE 4
Nemenyi Post-Hoc Tests for Comparing Each Pair of Algorithms' Average Rankings

Algorithms	Avg Rank Diff
Pop-EA vs (1+1) EA	**0.8958**
Pop-EA vs RLS	**1.5417**
Pop-EA vs (1+1) EA no-norm	**2.8125**
(1+1) EA vs RLS	0.6458
(1+1) EA vs (1+1) EA no-norm	**1.9167**
RLS vs (1+1) EA no-norm	**1.2708**

Differences of average ranks of at least the critical distance $CD = 0.6770$ are statistically significant at the level of $\alpha = 0.05$ and are highlighted in bold.

normalization, normalization plays an important role in improving the algorithm's performance. In addition, as the hit rate of the $(1+1)$ EA is the same as the hit rate of RLS and Pop-EA, the choice among these three algorithms with normalization did not alter the results in terms of hit rate for these problem instances.

Another interesting observation is that our algorithms with normalization always managed to achieve hit rate of 100 independent of the problem features. The previous study [6] revealed that the GA's hit rate varied depending on the problem features. Our experiments using $(1+1)$ EA without normalization also presented different hit rates for different instances. So, normalization also plays an important role in making the hit rate of the algorithm less dependent on the problem features. In other words, it helps to improve robustness of the algorithm to different problem instances. As the hit rate was the same for all algorithms with normalization, they all presented similar robustness.

In summary, this section shows that the algorithms with normalization obtained the perfect hit rate, for all tested problem instances.

7.4 Solution Quality

Another criterion that should be considered by a software manager when choosing a PSP algorithm is the quality of the solutions provided, i.e., how good the cost and completion time of the solutions is. In this section, we analyze the quality of the feasible solutions in terms of fitness, cost and completion time.

7.4.1 Ranking of Approaches According to Fitness

The fitness function used in this paper is a measure of the solution quality given the relative importance of cost and completion time selected by a project manager. Following Demšar's recommendation for comparing two algorithms over multiple data sets [42], we used the Friedman test to compare the average ranking of the $(1+1)$ EA, RLS, Pop-EA and the $(1+1)$ EA without normalization according to their fitness across problem instances. The average ranking is shown in Table 3. The GA has not been ranked here because we do not have its corresponding fitness values from [6]. The Friedman test detected statistically significant difference in the average ranks ($F_F = 246.30 > F(3, 141) = 2.67$, p-value less than 0.0001).

As the Friedman test detected statistically significant difference, the Nemenyi post-hoc test [42] was then used to compare the average rank of each algorithm against each other. In this way, it is possible to know which of the algorithms actually performed differently or similarly to each other. The test detected statistically significant differences between all algorithms at the level of $\alpha = 0.05$ except between $(1+1)$ EA and RLS (see Table 4). These results together with the average rankings show that Pop-EA obtained the best average rank across problem instances, $(1+1)$ EA without normalization obtained the worst, and $(1+1)$ EA and RLS obtained similar average ranks to each other.

It is worth noting that the rank obtained by the $(1+1)$ EA without normalization was always the worst, for all problem instances (the standard deviation of its ranking is zero in Table 3). Normalization was able to improve fitness, in such a way that our $(1+1)$ EA achieved better ranking than the $(1+1)$ EA without normalization. So, normalization played an important role to improve fitness. The use of a population through Pop-EA improved the fitness further.

Pop-EA obtained the best average ranking across problem instances. Out of 48 problem instances there were only five instances where Pop-EA was not the best ranked algorithm:

- benchmark 4's instance with 10 tasks, six-seven employees' skills, 10 employees;
- benchmark 4's instance with 10 tasks, four-five employees' skills, five employees;
- benchmark 5's instance with 10 tasks, five project skills, 10 employees;
- benchmark 5's instance with 20 tasks, 10 project skills, five employees;
- benchmark 5's instance with 20 tasks, 10 project skills, 15 employees.

For these problem instances, either the $(1+1)$ EA or RLS obtained the best rank. These instances have varied number of tasks, employees' skills, project skills and employees. It was not possible to identify a pattern that reveals for what type of instances the Pop-EA was not the best ranked algorithm. However, we show in Section 7.4.2 that the risk of using Pop-EA instead of $(1+1)$ EA or RLS is very small.

In summary, this section shows that normalization plays an important role in improving the $(1+1)$ EA's fitness, and that the use of Pop-EA was able to improve it further, whereas RLS obtained similar fitness to the $(1+1)$ EA. Pop-EA usually obtained the best fitness.

7.4.2 Practical Effect of the Differences in Fitness

In addition to the statistical significance of the difference in fitness, it is also important to analyze the practical

TABLE 5
Statistics of the Practical Effect (Equation 2) of Differences in Solution Quality across Problem Instances

	Best against worst fitness ranked algorithm with normalisation		Best fitness ranked algorithm against (1+1) EA without normalisation	
	Cost	Duration	Cost	Duration
Minimum PE	0.0000%	0.0086%	0.1682%	9.9823%
Maximum PE	0.0238%	0.3414%	3.3208%	174.1823%
Average PE	0.0025%	0.1068%	1.2640%	51.6686%
Std Deviation PE	0.0043%	0.0845%	0.8177%	39.7585%
Maximum difference in units of time and cost	172.8333	0.0479	57521.8065	94.0596

The actual maximum difference in units of cost and time is also shown. Statistics regarding the difference in cost do not consider problem instances with fixed cost (benchmarks 1-3), and only feasible solutions are considered.

TABLE 6
Minimum and Maximum Values of Standard Deviation in Cost and Completion Time across Different Solutions for the Same Problem Instance Divided by Average

	Pop-EA	(1+1) EA	RLS	(1+1) EA no-norm
Minimum stdev / avg cost	0.0000%	0.0010%	0.0059%	0.2537%
Maximum stdev / avg cost	0.0148%	0.0344%	0.0378%	1.9022%
Minimum stdev / avg completion time	0.0002%	0.0183%	0.0233%	4.6455%
Maximum stdev / avg completion time	0.2768%	0.1834%	0.5850%	52.1864%

Statistics regarding minimum standard deviation of cost do not consider the problem instances with fixed cost (benchmarks 1-3), and only statistics for problem instances with at least three feasible solutions are considered.

differences in solution quality when choosing an algorithm. For example, a project manager could still opt for using an algorithm likely to produce worse solution quality than the best one if the difference in quality between these two algorithms is unlikely to be high. A possible reason for a software manager to opt for a worse algorithm would be if he/she has easier access to the implementation of this algorithm or the algorithm is far more efficient to execute than others. If the software manager can use the algorithm most likely to be the best, he/she should still be aware of the risk incurred in this choice. For example, if an algorithm is likely to produce the best solution quality in most cases, and in a few cases the solution quality is likely to be just slightly worse than other algorithms, then the risk incurred in opting for this "best" algorithm is very small. Thus, it is reasonable for a software manager to always opt for it. On the other hand, if this algorithm is likely to perform much worse than another in a few cases, the risk in always choosing it is high.

We define the practical effect of differences in solution quality for a certain problem instance i as:

$$\mathrm{PE}_i = \frac{|\mathrm{measure}(A) - \mathrm{measure}(B)|}{\mathrm{measure}(A)}, \qquad (2)$$

where A is the algorithm with the best fitness for the problem instance i, B is an alternative algorithm, and measure is either the average cost or completion time. This is a measure of how large the difference in terms of cost/completion time is in relation to the cost/completion time of the best fit solution.

Table 5 presents the minimum, maximum, average and standard deviation of the practical effect in terms of cost and completion time across problem instances. We can observe that the practical effect in terms of cost and completion time considering only the set of algorithms with normalization is very low. For both cost and completion time, the maximum practical effect is considerably less than one percent. In contrast, the practical effects between the best ranked algorithm with normalization and the $(1+1)$ EA without normalization are much larger.

For illustration purposes, Table 5 also shows the actual maximum difference between the best and the worst fitness ranked algorithms in cost and time. Considering that the salary of an employee is around $10,000 per unit of time, let's regard units of time as months. We can see that the maximum difference in cost between the best and the worst fitness ranked algorithms with normalization is only approximately $173, whereas the difference considering the $(1+1)$ EA without normalization is of approximately $57,521. In terms of completion time, the differences are about one day and a half, and more than seven years, respectively. This further illustrates the practical effect of choosing between the algorithms with and without normalization.

In summary, this section shows that it is important to opt for algorithms with normalization in terms of practical effect of solution quality, and that the risk incurred in choosing one or another algorithm with normalization is not high. For instance, a software manager could opt for the Pop-EA, which has the best average rank in terms of fitness and is very unlikely to perform much worse than the other algorithms.

7.4.3 Variance of Cost and Completion Time

The variance in cost and completion time is another factor that should be taken into account when choosing an algorithm. If the solution quality when running an algorithm with different random seeds varies a lot, it would be difficult for the software manager to know whether a certain run is likely to have produced good or bad solutions.

The algorithms with normalization obtained very low variances in cost and completion time across solutions. Table 6 shows the minimum and maximum standard deviation divided by the average. Statistics regarding the GA were not available from [6]. The standard deviation in cost was never more than 0.04 percent of the average cost and the standard deviation in completion time was never more than 0.60 percent of the average completion time of the project for the algorithms with normalization. When normalization

TABLE 7
Quality of Feasible Solution Measured by the Cost Divided by the Completion Time

(a) Benchmark 1: avg. $\mathrm{cost/time}$. The optimal value is $n \cdot 10{,}000$.

n	Pop-EA	(1+1) EA	RLS	(1+1) EA no-norm	GA [6]
5	**49,998**	49,978	49,901	36,581	44,790
10	**99,981**	99,940	99,828	75,559	86,957
15	**149,964**	149,900	149,660	114,754	126,779
20	**199,919**	199,830	199,614	151,235	166,667

(b) Benchmark 2: avg. $\mathrm{cost/time}$. The optimal value is 50,000.

m	Pop-EA	(1+1) EA	RLS	(1+1) EA no-norm	GA [6]
10	**49,998**	49,978	49,901	36,579	44,944
20	**49,984**	49,980	49,964	35,273	44,748
30	**49,999**	49,990	49,955	18,236	–
Stdev	8.34	6.43	34.09	10,234.18	–

(c) Benchmark 3: avg. $\mathrm{cost/time}$. The optimal value is 50,000.

sk	Pop-EA	(1+1) EA	RLS	(1+1) EA no-norm	GA [6]
2	**49,998**	49,983	49,925	–	45,230
4	**49,996**	49,983	49,888	–	45,069
6	**49,998**	49,980	49,894	38,762	44,651
8	**49,998**	49,979	49,905	36,518	44,617
10	**49,998**	49,978	49,901	37,182	44,427
Stdev	0.71	2.30	13.95	–	336.18

The averages were calculated considering only the runs in which a feasible solution was found. The best values are in bold. n, m and sk are the number of employees, tasks and employees' skills. Stdev is the standard deviation of the average values reported in the table.

was not used in the $(1+1)$ EA, the variance in cost was from 0.25-1.90 percent of the total cost and the variance in completion time was 4.65-52.19 percent of the total completion time whenever there were more than three feasible solutions. So, considering the criterion variance, it would be good for a software manager to opt for algorithms with normalization.

The reason for the low variances is because normalization made it easier to find near optimal solutions, as shown in Section 7.4.4. As it was easier to find near optimal solutions, there were less cases where the solution deviated considerably from the optimum, and thus the variances were lower.

7.4.4 Distance to the Optimal Cost and Completion Time

No matter whether a certain algorithm produces solutions of better or worse quality than another, if all these solutions are far away from the best cost and completion time, i. e., if the cost and completion time of the solutions are very poor, then these algorithms are not useful. As we know the optimal cost and completion time for benchmarks 1-3, it is possible to evaluate how close the solutions found by the algorithms are to these optima. Tables 7 and 8 show some values descriptive of the solution quality for benchmarks 1-3. The cost for all solutions of benchmarks 1, 3 and the first instance of benchmark 2 were \$980,000. The cost for the second and third instances of benchmark 2 were \$2,600,000 and \$2,700,000, respectively.

The cost per unit of time ($\mathrm{cost/time}$) for benchmarks 2 and 3 is optimal at \$50,000 (all employees working full time), as there are five employees and all have the same salary. Each problem instance of benchmark 1 has the optimal $\mathrm{cost/time}$ increased by \$50,000 in relation to the previous instance, as the number of employees is increased by 5. As we can see from Table 7, all our

TABLE 8
Quality of Feasible Solution Measured by the Number of Employees (n) Multiplied by the Completion Time (time)

Benchmark 1: average $n \cdot \mathrm{time}$					
n	Pop-EA	(1+1) EA	RLS	(1+1) EA no-norm	GA [6]
5	**98.00**	98.04	98.19	113.96	109.40
10	**98.02**	98.06	98.17	129.68	112.70
15	**98.02**	98.07	98.22	128.06	115.95
20	**98.04**	98.08	98.19	129.54	117.60
Stdev	0.01	0.02	0.02	7.60	3.63

The averages were calculated considering only the runs in which a feasible solution was found The best values are in bold. Stdev is the standard deviation of the average values reported in the table.

algorithms with normalization obtained near-optimal solutions for these benchmarks, and our Pop-EA obtained solutions even closer to the optimum than the other algorithms with normalization. The algorithms without normalization, in contrast, obtained solutions further away from the optimum.

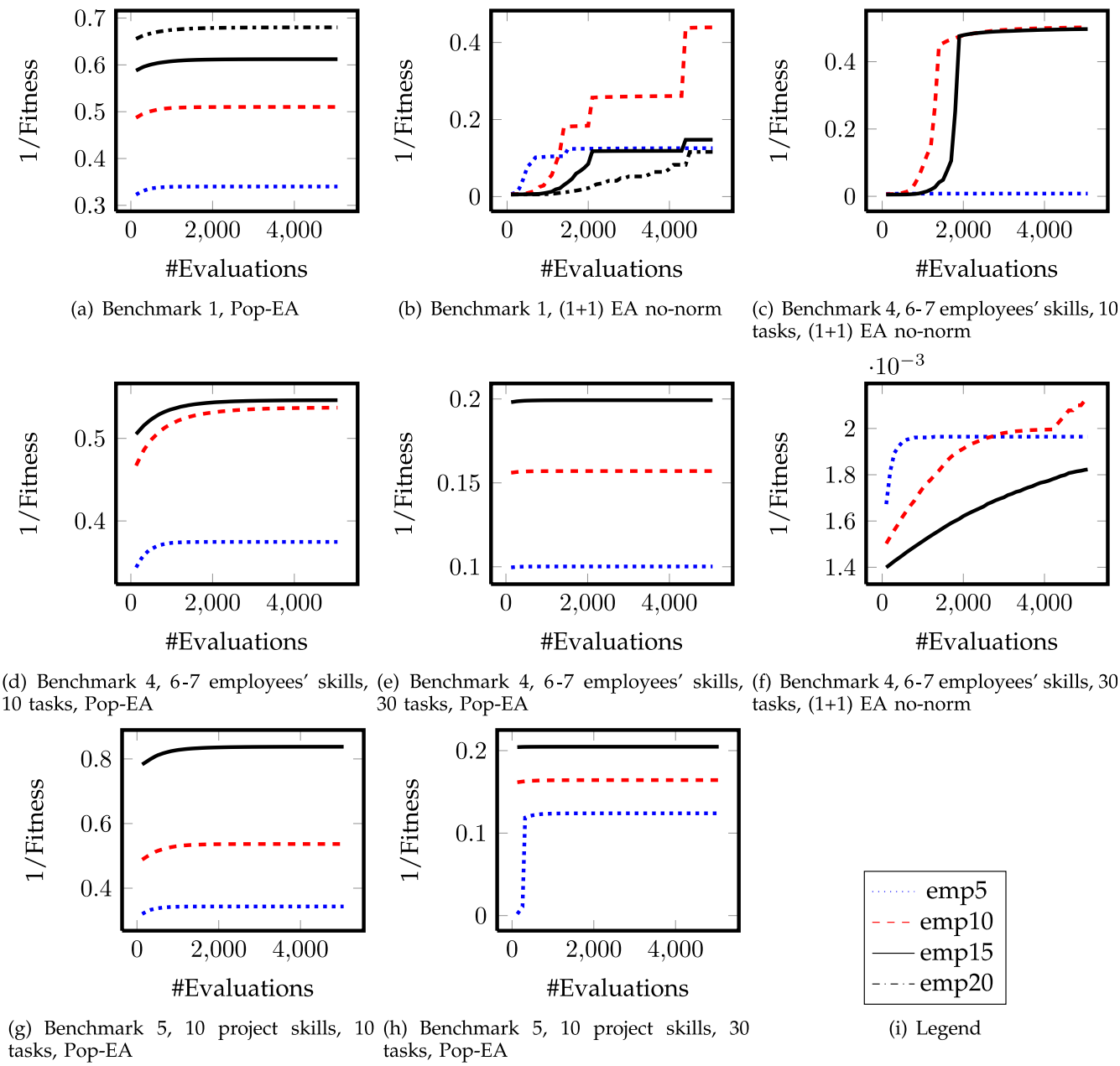
It is worth noting that these results corroborate the analysis presented in Section 7.4.1, which show that normalization plays a key role in improving the solution quality obtained by the $(1+1)$ EA, and that Pop-EA manages to improve it further through the use of a population. They also corroborate the results in Section 7.4.2, which show that the practical effect of using an algorithm with normalization instead of without normalization was large, whereas choosing among the three algorithms with normalization did not change solution quality as much.

The results also show that the solution quality of our algorithms with normalization were less affected by variations in the number of employees, tasks and employees' skills than the other algorithms'. This is shown by their more similar product $n \cdot \mathrm{time}$ across instances of benchmark 1 (lower standard deviation in Table 8), and by their more similar $\mathrm{cost/time}$ across instances of benchmarks 2 and 3 (lower standard deviation in (Tables 7b and 7c). These values were more different across instances of a same benchmark for the algorithms without normalization.

In summary, this section shows that the solutions found by the algorithms with normalization for benchmarks 1-3 were near-optimal, whereas the solutions found by the $(1+1)$ EA without normalization and by the GA were further from the optimal cost and completion time.

7.5 Evaluation of Convergence Time

This section complements the analysis of how much each algorithm is influenced by the features of the problem instances such as the number of employees and tasks. If the solution quality varies a lot depending on the features of the problem instance, then the choice for an algorithm is more difficult and incurs more risk. Sections 7.3 and 7.4 show that the hit rate and the quality of the solutions produced by algorithms with normalization were much less affected by problem features than the ones produced by the algorithms without normalization. However, even if the hit rate and the solution quality obtained by a certain algorithm for different problem instances are not affected by different features of these instances, that does not necessarily mean that the algorithm is not affected by such features.



(a) Benchmark 1, Pop-EA

(b) Benchmark 1, (1+1) EA no-norm

(c) Benchmark 4, 6-7 employees' skills, 10 tasks, (1+1) EA no-norm

(d) Benchmark 4, 6-7 employees' skills, 10 tasks, Pop-EA

(e) Benchmark 4, 6-7 employees' skills, 30 tasks, Pop-EA

(f) Benchmark 4, 6-7 employees' skills, 30 tasks, (1+1) EA no-norm

(g) Benchmark 5, 10 project skills, 10 tasks, Pop-EA

(h) Benchmark 5, 10 project skills, 30 tasks, Pop-EA

(i) Legend

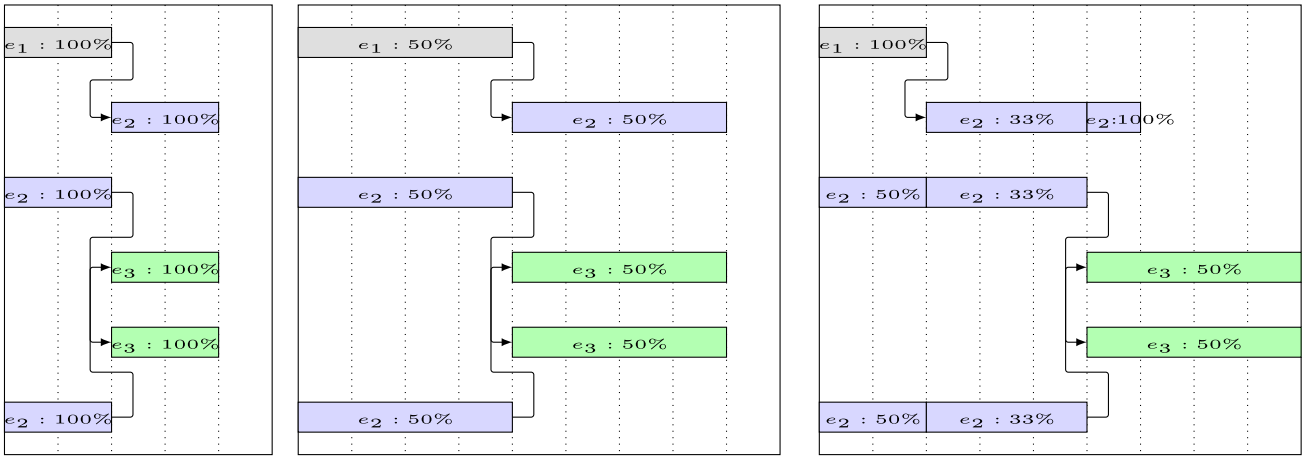
Fig. 4. Convergence plots.

An analysis of convergence time can show to what extent the algorithms were affected by problem features, providing useful insight into whether the algorithm would still be able to provide good solutions if the problem grew a bit more. Fast convergence to good solutions means that if the problem was a bit larger, the algorithm may still be able to find good solutions within the budget of fitness evaluations.

We analyze the convergence plots of the algorithms with the best and the worst fitness ranking: Pop-EA and $(1+1)$ EA without normalization, respectively (Section 7.4.1). The plots were generated for each problem instance and use the inverse of the average fitness across multiple runs, rather than the fitness itself. This allows extra visual comparison of the plots of the Pop-EA against the plots of the GA published in [6]. These plots are comparable because the inverse of the fitness function of the Pop-EA is the same as the fitness function of the GA when the solutions are feasible, and Pop-EA obtained feasible solutions in 100 percent of the runs.

Examples of representative plots are shown in Fig. 4. Pop-EA frequently starts with solutions of considerably good fitness, whereas the algorithms without normalization (both $(1+1)$ EA without normalization and the GA) typically start with solutions of very low quality (1/fitness close to zero). For example, Pop-EA's inverse of fitness in Figs. 4a, 4d, and 4e starts higher than the $(1+1)$ EA no-norm's in Figs. 4b, 4c, and 4f, respectively. This is because normalization eases the overwork constraint, allowing for feasible solutions to be found very quickly, whereas the algorithms without normalization need longer time to find feasible solutions.

In addition to frequently starting off with solutions of better quality, the Pop-EA also converges very quickly, usually within 1,280 fitness evaluations (20 generations). While



(a) Infeasible schedule (b) Schedule after repair (c) Schedule with normalisation

Fig. 5. An example of a schedule where repair leads to shorter completion times than normalization. Dedication values of three employees e_1, e_2, e_3 are shown; all dedications not listed are 0. Left: an infeasible schedule with overwork. Middle: the same schedule after repair. Right: the same schedule with normalization.

algorithms without normalization struggled to find feasible solutions, Pop-EA was able not only to quickly find feasible solutions, but also to converge faster to solutions of better quality. This occurs both for benchmarks 1-3, for which it is known that Pop-EA found near-optimal solutions (Section 7.4.4), and for benchmarks 4 and 5. Even though we do not know the optimal cost and duration for benchmarks 4 and 5, the fact that convergence was faster for all benchmarks and the better fitness achieved by the Pop-EA in comparison to $(1+1)$ EA without normalization suggest that this fast convergence has led to solutions of good, possibly near-optimal, quality, rather than being a premature convergence to poor solutions.

The convergence time usually did not present considerable changes for instances with different numbers of employees, employee's skills and project skills. For example, the convergence time is similar for different numbers of employees in Fig. 4a. This confirms that Pop-EA was not much affected by changes in these features for these problem instances. Changes in the number of tasks affected the convergence time a bit more, with a tendency of smaller numbers of tasks taking longer time to converge. This behavior is exemplified by Fig. 4d versus Fig. 4e and Fig. 4g versus Fig. 4h. This is a somewhat surprising behavior, as one would expect instances with more tasks to be more difficult and the algorithm to take longer to converge. However, as the difference between the final and optimal solution quality from 10 to 20 and 30 tasks in benchmark 2 (Table 7b) does not decrease monotonically, it is possible that the difference in difficulty of these instances is due to random factors involved in the instance generation. Note that instances with different numbers of tasks have different randomly generated TPGs, which may considerably change the difficulty of the problem.

In summary, this section shows that Pop-EA managed to quickly find feasible solutions; it presented generally lower convergence time than the $(1+1)$ EA without normalization and the GA; and its convergence time was not much affected by features such as numbers of employees, employee's skills and project skills.

8 Discussion

Our theoretical and empirical results have confirmed that normalization is very effective in avoiding overwork, and that EAs with normalization quickly and consistently evolve schedules of good quality. However, one question remains open: is normalization better than repair? More precisely, is the completion time with normalization always better than with repair?

It is tempting to think that the answer to the latter is *yes*. After all, repair divides *all* tasks by the maximum overwork, across the whole schedule, whereas normalization only divides dedications where necessary. Intuitively, since normalization divides dedications by less, normalized schedules should be no longer than repaired ones. If x is a feasible schedule, $\mathrm{time}(\mathrm{repair}(x))$ denotes the completion time of the repaired schedule x, and $\mathrm{time}(\mathrm{normalize}(x))$ denotes the completion time of the normalized schedule x, we might think that for all instances and all feasible x

$$\mathrm{time}(\mathrm{normalize}(x)) \leq \mathrm{time}(\mathrm{repair}(x)).$$

However, it turns out that this is not true for all schedules. Fig. 5 gives an example where, for a particular configuration of dedication values, repair leads to a shorter completion time than normalization. In this example repair doubles all execution times. Compared to this schedule, normalization allows e_1 to finish his/her job earlier. This means that, because of the precedence constraints, e_2 starts to work on three tasks simultaneously. The bottom two tasks e_2 is working on therefore finish later, and this postpones the time e_3 is allowed to start on her/his tasks. Overall, the completion time with repair is 8, and the completion time with normalization is 9.

Theorem 12. *There exists a PSP instance and a feasible solution x such that*

$$\mathrm{time}(\mathrm{normalize}(x)) > \mathrm{time}(\mathrm{repair}(x)).$$

So we cannot claim that normalization is *always* better than repair, with regard to the completion time only.

However, what can be said is that the optimal completion time with normalization is never more than the optimal completion time with repair. This is because the all-ones dedication matrix has every employee always working full time, leading to a minimal completion time (cf. Theorem 2).

Theorem 13. *For every solvable PSP instance the shortest completion time of any feasible schedule with normalization is no larger than the shortest completion time of any feasible schedule with repair. Formally, if X is the set of all feasible schedules,*

$$\min_{x\in X}\{\text{time}(\text{normalize}(x))\} \leq \min_{x\in X}\{\text{time}(\text{repair}(x))\}.$$

The disadvantage of using normalization is that schedules become more complex as the dedication for particular jobs may change as other jobs are started or finished.

9 Conclusions

Research in search-based software engineering has primarily been experimental so far. Few theoretical studies exist. This paper makes a concerted effort, through both theoretical analysis and experimental studies, in understanding more deeply why and when an EA works (or fails) on some PSP problems. The theoretical analysis has inspired new designs of EAs, which have been shown to perform better than existing work.

The new design includes normalization of dedication values, a tailored mutation operator, and fitness functions with a strong gradient towards feasible solutions. Normalization removes the problem of overwork and allows an EA to focus on the solution quality. It facilitates finding the right balance between dedication values for different tasks and allows employees to adapt their workload whenever other tasks are started or finished. This is an advantage over the repair mechanism [7] which decreases dedications uniformly for all tasks and at all times.

We have derived general upper and lower bounds for the expected optimization time of broad classes of EAs. More specific results were presented for RLS and the $(1+1)$ EA, including a general upper bound on the time a feasible solution is reached. For linear schedules both the $(1+1)$ EA and RLS are effective. However, despite using normalization they still struggle to escape from local optima where many dedication values form an equilibrium.

Our empirical study is based on comparisons among three algorithms with normalization ($(1+1)$ EA, RLS and Pop-EA) and two algorithms without normalization ($(1+1)$ EA no-norm and a state-of-the-art GA). Our analyzes confirm that normalization is very effective in improving solutions in terms of several criteria (hit rate, solution quality and convergence time) that may influence a software manager's decision in adopting a certain algorithm. Our results show that algorithms with normalization should be favored over algorithms without normalization when considering these criteria, and that normalization makes the EAs more robust to different problem instances. The practical effect of the differences in solution quality between algorithms with and without normalization is high, whereas the differences among the algorithms with normalization are small, with the Pop-EA achieving the best results.

Future work includes experimental analysis of the runtime or generation-to-success distributions [43], [44], empirical comparison against the repair operator, case studies using real-world software projects, and investigation of different problem formulations (e. g., when different combinations of dedications affect the productivity). The solution quality in single-objective EAs such as the ones investigated in this work is inherent to the fitness, which considers the relative importance between cost and completion time based on their weights. We shall propose the use of other weights for the fitness function and the extension of our design to multi-objective formulations of the problem.

Appendix

This appendix contains proofs omitted from the main part.

Proof of Lemma 1. If y has missing skills then the claim is obvious as then both cost and completion times are proportional to the number of missing skills, reqsk.

Now assume that y has no missing skills and x has reqsk missing skills. The completion time for any feasible schedule is bounded by $k\sum_{j=1}^{m}\text{eff}_j$. This corresponds to a schedule where all tasks are processed sequentially and for each task there is only one employee working on it, with the minimum dedication of $1/k$. Paying all employees their full salary during the time needed for one employee to work full time yields an upper bound of $\sum_{i=1}^{n}\sum_{j=1}^{m}s_i\text{eff}_j$ on the costs. Those upper bounds for completion times and costs are lower than all penalties for infeasible solutions, hence we have

$$\text{time}(y) \leq k\sum_{j=1}^{m}\text{eff}_j \;<\; \text{reqsk}\cdot 2k\sum_{j=1}^{m}\text{eff}_j = \text{time}(x),$$

as well as

$$\text{cost}(y) \leq \sum_{i=1}^{n}\sum_{j=1}^{m}s_i\text{eff}_j \;<\; \text{reqsk}\cdot 2\sum_{i=1}^{n}\sum_{j=1}^{m}s_i\text{eff}_j = \text{cost}(x).$$

□

Proof of Theorem 3. Let μ be the number of solutions in the initial population. For very large populations, $\mu > knm\log(nm))$, we argue as follows: the probability that a uniform random solution hits the target point is $(k+1)^{-nm}$. Hence, the probability that one of the first $knm\log(nm)$ constructed solutions hits the target point is at most $knm\log(nm)\cdot(k+1)^{-nm}$. Hence, with probability at least $\Omega(1)$ this does not happen, and this gives a lower bound of $\Omega(1)\cdot knm\log(nm)$ on the expectation as claimed. In the following, we assume $\mu \leq knm\log(nm)$.

Call an entry of the dedication matrix *bad* if it disagrees with the global optimum. Observe that in order to find the optimum it is necessary to change all bad entries at least once in a mutation. For an initial solution created uniformly at random, each entry is bad at initialization with probability $k/(k+1)$. The expected number of bad initial entries is $knm/(k+1)$. By classical Chernoff bounds [45], with probability $1-e^{-\Omega(nm)}$ the initial number of bad entries is at least $nm/3$. The probability that all of these will have at least $nm/3$ bad entries is at least $1-\mu\cdot e^{-\Omega(nm)} = 1-e^{-\Omega(nm)+\log(\mu)} = 1-e^{-\Omega(nm)}$ by the union bound.

Since both local and global mutations treat all matrix entries in the same fashion, we can assume without loss of generality that all initial solutions share the same $nm/3$ entries in the matrix which are bad. Then a necessary condition for finding the target is that each of these entries is turned good at least once in at least one individual.

Let $t := (knm-1)\ln(nm/3)$. The probability of not changing a particular entry in t mutations is $(1-1/(nm))^t$. The probability that there is a bad entry which is never turned good during t mutations is at least

$$\begin{aligned}
&1-\left(1-\left(1-\frac{1}{knm}\right)^t\right)^{nm/3}\\
&\geq 1-\left(1-e^{-\ln(nm/3)}\right)^{nm/3}\\
&\geq 1-\left(1-\frac{3}{nm}\right)^{nm/3}\\
&\geq 1\ -e^{-1},
\end{aligned}$$

where in the last step we have used $(1-1/x)^x \leq e^{-1}$ for $x \geq 1$. Using the union bound for the initialization, with probability at least $1-e^{-1}-e^{-\Omega(nm)} = \Omega(1)$ the EA has not found an optimum after $t = \Omega(knm \cdot \ln(nm))$ steps. This establishes the bound $\Omega(1)\cdot t = \Omega(knm \cdot \log(nm))$. □

Proof of Theorem 4. With every constructed solution, global mutation is applied to a parent or an intermediate solution resulting from recombination or other operators. Call this solution x and fix a global optimum x^*. If x and x^* differ in ℓ dedication entries, the probability of global mutation turning x into x^* is exactly

$$\left(\frac{1}{knm}\right)^{\ell}\cdot\left(1-\frac{1}{nm}\right)^{nm-\ell} \geq \left(\frac{1}{knm}\right)^{nm}$$

as the probability of mutating a wrong entry of the dedication matrix and selecting the right one is $1/(nm)\cdot 1/k$. The expected waiting time for an event that happens with probability at least $(knm)^{-nm}$ in each time step is at most the inverse probability, $(knm)^{nm}$. □

In order to prove Theorem 7, we use the following general result about separable functions (Theorem 12 in [38], partly extended as described in Corollary 13 [38]).

Theorem 14 (Doerr et al. [38]). *Let $f = \sum_{i=1}^{m} w_i f_i$ where $w_i \in \mathbb{Z}$ and all f_i are defined on pairwise disjoint sets of variables. For each $1 \leq i \leq n$, let f_i fulfill the following assumptions. There are $\ell_i \in \mathbb{N}$ and $0 = a_{i0} < a_{i1} < \cdots < a_{i\ell_i} = \max\{f_i(x) \mid x \in S\}$. For all $1 \leq j \leq \ell_i$, let $d_{ij} := a_{ij} - a_{ij-1}$. Assume that $d_{ij} \geq d_{ij+1}$ for all $1 \leq j < \ell_i - 1$. Let $A_{ij} := \{x \in S \mid a_{ij-1} \leq f_i(x) < a_{ij}\}$ for all $1 \leq j \leq \ell_i + 1$, where $a_{i\ell_i+1} := \infty$. Assume that for all $1 \leq j \leq \ell_i$ and all $x \in A_{ij}$, the $(1+1)$ EA optimizing f_i with current search point equal to x with probability p in one iteration finds a search point y with $f_i(y) \geq f_i(x) + d_{ij}$.*

Then the expected optimization time of the $(1+1)$ EA on f is at most $e(1/p)\sum_{i=1}^{m}\ell_i$.

Proof of Theorem 7. The time for reaching feasibility is smaller than the time claimed, according to Theorem 5, so we only need to consider the expected time after a feasible schedule has been found.

For linear schedules with a feasible solution x the completion time is a separable function in a sense defined above:

$$\mathrm{time}(x) = \sum_{i=1}^{m} \mathrm{eff}_i \cdot \mathrm{time}_i(x),$$

where $\mathrm{time}_i(x) = 1/\sum_{j=1}^{n} x_{j,i}$ is the unweighted completion time for the i-th task. Note that the optimal value of $\mathrm{time}_i(x)$ is $1/n$, when all employees have dedication 1 on task i. Since Theorem 14 is defined for maximization with the minimum fitness value for each subfunction being at 0, an equivalent problem is maximizing $f(x) = \sum_{i=1}^{m}\mathrm{eff}_i \cdot f_i(x)$ where $f_i(x) = k - \mathrm{time}_i(x)$. Note that all f_i concern mutually disjoint sets of dedication values, and f_i attains its minimum at 0, when only one employee has a positive dedication of $1/k$.

In the notation of Theorem 14 for each $1 \leq i \leq m$ we have $\ell_i := nk$ different f_i-values $a_{i0} < a_{i1} < \cdots < a_{ink-1}$ with

$$a_{ij} = k - \frac{1}{(j+1)/k} = k\cdot\frac{j}{j+1}.$$

The differences between subsequent a_{ij}-values are decreasing: for $d_{ij} := a_{ij} - a_{ij-1}$ and all $1 \leq j < n$ we have

$$\begin{aligned}
d_{ij} &= k\left(\frac{j}{j+1} - \frac{j-1}{j}\right)\\
&= \frac{k}{j(j+1)} \geq \frac{k}{(j+1)(j+2)} = d_{ij+1}.
\end{aligned}$$

Now, if the $(1+1)$ EA was optimizing f_i as fitness function, it could increase the current f_i-value by increasing the dedication of any employee that does not yet have dedication 1. This happens with probability at least $1/(knm)\cdot(1-1/(nm))^{nm-1} \geq 1/(eknm)$. By Theorem 14, the expected optimization time of the $(1+1)$ EA on f is therefore at most $e\cdot eknm\cdot\sum_{i=1}^{m} nk = O((knm)^2)$. □

For the proof of Theorem 9 we use the following theorem on separable pseudo-Boolean functions, i. e., a separable function defined on the space $\{0,1\}^{nm}$. This is the case here as we assume $k = 1$.

The following theorem states that, roughly speaking, under certain conditions the $(1+1)$ EA can optimize subfunctions of a separable function in parallel, with a mild overhead of a logarithmic factor.

Theorem 15 (Doerr et al. [38]). *Let $f = \sum_{i=1}^{k} w_i f_i$ where $w_i \in \mathbb{Z}$ and each $f_i : \{0,1\}^{\ell} \to \{0,1,\ldots,r\}$ attains integer values between 0 and $r \in \mathbb{N}$. Suppose f is separable, i. e., there are mutually disjoint sets $I_1,\ldots,I_k \in \{1,\ldots,\ell\}$ such that f_i only depends on bits in I_i.*

Assume $r = o(\log^{1/2}\ell)$ and let T^* be an upper bound on the expected optimization time of the $(1+1)$ EA that holds on every f_i, $1 \leq i \leq k$, for every initial search point. Then the expected optimization time of the $(1+1)$ EA on f is at most $(1+o(1))(4e^2 rT^*\ln\ell)$.

Proof of Theorem 9. Since all salaries are equal, the costs for the project are fixed. Hence with every Pareto-compliant fitness function the $(1+1)$ EA behaves like it was minimizing the completion time. The completion time can be written as a function $f = \sum_{i=1}^{m} w_i f_i$ where $w_i f_i$ denotes

the completion time of the i-th task. This holds since all tasks have to be processed sequentially. Taking $w_i = \text{eff}_i$, this leaves f_i reflecting the inverse of the total dedication of all employees.

Considering the $(1+1)$ EA with mutation probability $1/(nm)$ on only one task, we show that with $n = O(1)$ we have $T^* = O(m)$ as an upper bound on the expected time until the completion time for this task is minimized, regardless of the initial solution. This is because the probability of decreasing any non-optimal completion time is at least $1/(nmk) \cdot (1-1/(nm))^{m-1} \geq \Omega(1/n)$ and $O(1)$ decreases suffice to reach a minimum completion time.

We cannot directly apply Theorem 15 since f_i may attain non-integral values. However, we know that f_i attains only n values in $\{1/n, 1/(n-1), 1/(n-2), \ldots, 1\}$. Considering $f_i' := n! \cdot f_i$ transforms these values into $\{n!/n, n!/(n-1),\ n!/(n-2), \ldots, n!\}$, all of which are integral. Also note that $n = O(1)$ implies $n! = O(1)$. Considering $w_i' := \text{eff}_i \cdot \frac{1}{n!}$ and applying Theorem 15 to $f = w_i' f_i'$ with $r = n! = O(1)$ and $\ell = nm$ yields an upper bound on the expected optimization time of

$$(1+o(1))(4e^2 \cdot n! \cdot T^* \ln(nm)) = O(m \log m).$$

□

Proof of Theorem 11. Consider the generalized instance described in Section 6.5. The global optimum is to set all dedications to 1, leading to a completion time of $\sum_{j=1}^{m} \text{eff}_j = 3m^2 + 1$. The solution where $d_{1,j} = 1/m$ for all $1 \leq j \leq m$ leads to a completion time of $3m^2 + m$; after $3m^2$ steps the first $m-1$ tasks are completed and then a remaining effort of 1 for the last one is completed with dedication $1/m$.

We claim that every other solution having at least one dedication value $1/m$ has a strictly worse completion time. If the $(1+1)$ EA starts with all dedication values set to $1/m$ then in order to reach a better solution, all entries need to be changed in one mutation. The probability of mutating all entries is $(nm)^{-nm} = m^{-m}$. The expected time until mutation escapes from this local optimum is therefore at least m^m, and this establishes a lower bound on the expected optimization time.

For proving the claim, we consider a modified instance where we redefine $\text{eff}_m := 3m$, so that all tasks have equal efforts. Since we have only decreased the effort of tasks, and there are no precedence constraints, the completion time for the original instance can only be larger.

Now assume a solution with ℓ dedication entries equal to $1/m$. All solutions with dedication values 0 are infeasible, hence we can assume that all other $m - \ell$ dedications are greater than $1/m$. As $k = m$, the next dedication value larger than $1/m$ is twice as large as $1/m$. Hence all tasks with dedication $1/m$ take at least twice as long as all other tasks. Therefore, there will be a time span of at least $3m^2/2$ time steps where only the latter ℓ tasks will be executed. During this time, the employee only works with total dedication ℓ/m, i. e., he/she is idle during a $(m-\ell)/m \geq 1/m$-fraction of the time. This amounts to a total idle time of at least $3m/2$. As the total workload that needs to be completed is $3m^2$, the completion time is at least

$$3m^2 + 3m/2 > 3m^2 + m.$$

This proves the claim for the modified and the original instance and the proof is complete. □

ACKNOWLEDGMENTS

The authors thank Enrique Alba for providing instances from [6], Francisco Chicano for pointing them to [7], Rami Bahsoon for the fruitful discussions, and the Associate Editor and anonymous reviewers for their constructive comments. This research was supported by EPSRC grant EP/J017515/1 and NSFC Grant 61329302. Xin Yao was supported by a Royal Society Wolfson Research Merit Award. A preliminary version with parts of the results and weaker theoretical results appeared in [1].

REFERENCES

[1] L.L. Minku, D. Sudholt, and X. Yao, "Evolutionary Algorithms for the Project Scheduling Problem: Runtime Analysis and Improved Design," *Proc. 14th Int' l Conf. Geneticand Evolutionary Computation Conf. (GECCO 12)*, pp. 1221-1228, 2012.

[2] T.K. Abdel-Hamid and S.E. Madnick, "The Dynamics of Software Project Scheduling," vol. 26, no. 5, pp. 340-346, 1983.

[3] I. Sommerville, *Software Engineering*. Addison-Wesley, 2001.

[4] M. Di Penta, M. Harman, and G. Antoniol, "The Use of Search-Based Optimization Techniques to Schedule and Staff Software Projects: An Approach and an Empirical Study," *Software: Practice and Experience*, vol. 41, no. 5, pp. 495-519, 2011.

[5] C.K. Chang, M.J. Christensen, and T. Zhang, "Genetic Algorithms for Project Management," *Annals of Software Eng.*, vol. 11, pp. 107-139, 2001.

[6] E. Alba and J.F. Chicano, "Software Project Management with GAs," *Information Sciences*, vol. 177, pp. 2380-2401, 2007.

[7] F. Luna, D. González-Álvarez, F. Chicano, and M.A. Vega-Rodríguez, "On the Scalability of Multi-Objective Metaheuristics for the Software Scheduling Problem," *Proc. 11th Int'l Conf. Intelligent System Design and Applications (ISDA '11)*, pp. 1110-1115, 2011.

[8] N. Nan and D. Harter, "Impact of Budget and Schedule Pressure on Software Development Cycle Time and Effort," *IEEE Trans. Software Eng.*, vol. 35, no. 5, pp. 624-637, Sept./Oct. 2009.

[9] E. Yourdon, *Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible"*. Prentice Hall PTR, 1997.

[10] K.J. Stewart and S. Gosain, "The Impact of Ideology on Effectiveness in Open Source Software Development Teams," *MIS Quarterly*, vol. 30, no. 2, pp. 291-314, 2006.

[11] B. Fitzgerald, "The Transformation of Open Source Software," *MIS Quarterly*, vol. 30, no. 3, pp. 587-598, 2006.

[12] N. Levina and J.W. Ross, "From the Vendor's Perspective: Exploring the Value Proposition in Information Technology Outsourcing," *MIS Quarterly*, vol. 27, no. 3, pp. 331-364, 2003.

[13] B. Ives and S.L. Jarvenpaa, "Applications of Global Information Technology: Key Issues for Management," *MIS Quarterly*, vol. 15, no. 1, pp. 33-50, 1991.

[14] W.-N. Chen and J. Zhang, "Ant Colony Optimization for Software Project Scheduling and Staffing with an Event-Based Scheduler," *IEEE Trans. Software Eng.*, vol. 39, no. 1, pp. 1-17, Jan. 2013.

[15] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proc. Future of Software Eng.*, pp. 342-357, 2007.

[16] A.E. Eiben and J.E. Smith, *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[17] C. Chao, "Spmnet: A New Methodology for Software Management," PhD dissertation, The Univ. of Illinois at Chicago, 1995.

[18] P. Kapur, A. Ngo-The, G. Ruhe, and A. Smith, "Optimized Staffing for Product Releases and Its Application at Chartwell Technology," *J. Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 365-386, 2008.

[19] C.K. Chang, H. Jiang, Y. Di, D. Zhu, and Y. Ge, "Time-Line Based Model for Software Project Scheduling with Genetic Algorithms," *Information and Software Technology*, vol. 50, no. 11, pp. 1142-1154, 2008.

[20] F. Chicano, F. Luna, A. J. Nebro, and E. Alba, "Using Multi-Objective Metaheuristics to Solve the Software Project Scheduling Problem," *Proc. 13th Ann. Geneticand Evolutionary Computation Conf. (GECCO '11)*, ACM, pp. 1915-1922, 2011.
[21] S. Gueorguiev, M. Harman, and G. Antoniol, "Software Project Planning for Robustness and Completion Time in the Presence of Uncertainty Using Multi Objective Search Based Software Engineering," *Proc. 11th Ann. Geneticand Evolutionary Computation (GECCO '09)*, pp. 1673-1680, 2009.
[22] J. Ren, M. Harman, and M. Penta, "Cooperative Co-Evolutionary Optimization of Software Project Staff Assignments and Job Scheduling," *Proc. Third Int'l Conf. Search Based Software Eng.*, vol. 6956, pp. 127-141, 2011.
[23] B. Boehm, *Software Engineering Economics*. Prentice-Hall, 1981.
[24] M. Shepperd and C. Schofield, "Estimating Software Project Effort Using Analogies," *IEEE Trans. Software Eng.*, vol. 23, no. 12, pp. 736-743, Nov. 1997.
[25] S. Chulani, B. Bohem, and B. Steece, "Bayesian Analysis of Empirical Software Engineering Cost Models," *IEEE Trans. Software Eng.*, vol. 25, no. 4, pp. 573-583, July/Aug. 1999.
[26] B. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D.J. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*. Prentice-Hall, 2000.
[27] E. Kocaguneli, T. Menzies, A. Bener, and J.W. Keung, "Exploiting the Essential Assumptions of Analogy-Based Effort Estimation," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 425-438, Mar./Apr. 2012.
[28] L.L. Minku and X. Yao, "Software Effort Estimation as a Multi-Objective Learning Problem," *ACM Trans. Software Eng. Methodology*, vol. 22, no. 4, p. 32, 2013.
[29] L.A. Maciaszek and B.L. Liong, *Practical Software Engineering—A Case Study Approach*. Addison-Wesley, 2005.
[30] A. Barreto, M. de O. Barros, and C. Werner, "Staffing a Software Project: A Constraint Satisfaction and Optimization-Based Approach," *Computers & Operations Research*, vol. 35, pp. 3073-3089, 2008.
[31] R. Kolisch and S. Hartmann, "Experimental Investigation of Heuristics for Resource-Constrained Project Scheduling: An Update," *European J. Operational Research*, vol. 174, no. 1, pp. 23-37, 2006.
[32] S. Droste, T. Jansen, and I. Wegener, "On the Analysis of the $(1+1)$ Evolutionary Algorithm," *Theoretical Computer Science*, vol. 276, no. 1/2, pp. 51-81, 2002.
[33] P.S. Oliveto, J. He, and X. Yao, "Time Complexity of Evolutionary Algorithms for Combinatorial Optimization: A Decade of Results," *Int'l J. Automation and Computing*, vol. 4, no. 3, pp. 281-293, 2007.
[34] F. Neumann and C. Witt, *Bioinspired Computation in Combinatorial Optimization—Algorithms and Their Computational Complexity*. Springer, 2010.
[35] *Theory of Randomized Search Heuristics—Foundations and Recent Developments*, vol. 1, A. Auger, B. Doerr, eds. World Scientific, 2011.
[36] D. Sudholt, "A New Method for Lower Bounds on the Running Time of Evolutionary Algorithms," *IEEE Trans. Evolutionary Computation*, vol. 17, no. 3, pp. 418-435, June 2013.
[37] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, second ed. MIT Press, 2001.
[38] B. Doerr, D. Sudholt, and C. Witt, "When Do Evolutionary Algorithms Optimize Separable Functions in Parallel?" *Proc. 12th Workshop Foundations of Genetic Algorithms (FOGA 2013)*, pp. 51-64, 2013.
[39] B. Doerr, T. Jansen, D. Sudholt, C. Winzen, and C. Zarges, "Mutation Rate Matters Even When Optimizing Monotonic Functions," *Evolutionary Computation*, vol. 21, no. 1, pp. 1-21, 2013.
[40] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4J—A Modular Framework for Meta-Heuristic Optimization," *Proc. Genetic and Evolutionary Computing Conf. (GECCO '11)*, pp. 1723-1730, 2011.
[41] A. Agresti and B. Coull, "Approximate Is Better than 'Exact' for Interval Estimation of Binomial Proportions," *The Am. Statistician*, vol. 52, pp. 119-126, 1998.
[42] J. Demšar, "Statistical Comparisons of Classifiers over Multiple Data Sets," *J. Machine Learning Research*, vol. 7, pp. 1-30, 2006.
[43] H. Hoos and T. Stützle, "Local Search Algorithms for SAT: An Empirical Evaluation," *J. Automated Reasoning*, vol. 24, no. 4, pp. 421-481, 2000.
[44] D. Barrero, B. Castaño, M. R-Moreno, and D. Camacho, "Statistical Distribution of Generation-to-Success in GP: Application to Model Accumulated Success Probability," *Proc. 14th European Conf. Genetic Programming (EuroGP '11)*, pp. 155-166, 2011.
[45] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge Univ. Press, 1995.

Leandro L. Minku received the BSc, MSc, and PhD degrees in computer science from the Federal University of Parana, Brazil, in 2003, the Federal University of Pernambuco, Brazil, in 2006, and the University of Birmingham, United Kingdom, in 2011, respectively. He is a research fellow at the Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), School of Computer Science, the University of Birmingham, United Kingdom. He was an intern at Google Zurich for six months in 2009/2010, received the Overseas Research Students Award (ORSAS) from the British government, and several scholarships from the Brazilian Council for Scientific and Technological Development (CNPq). His main research interests include search-based software engineering, software prediction models, machine learning in changing environments, and ensembles of learning machines. His work has been published in internationally renowned journals such as *ACM Transactions on Software Engineering and Methodology*, *IEEE Transactions on Knowledge and Data Engineering*, and *Neural Networks*.

Dirk Sudholt received the Diplom (master's) degree in 2004 and the PhD degree in computer science in 2008 from the Technische Universität Dortmund, Germany, under the supervision of Prof. Ingo Wegener. He is a lecturer in computer science at the University of Sheffield, United Kingdom. He has held postdoc positions at the International Computer Science Institute, University of California at Berkeley, California, and at the University of Birmingham. His research focuses on the optimization time of randomized search heuristics such as evolutionary algorithms, in particular memetic and parallel variants, and swarm intelligence algorithms like ant colony optimization and particle swarm optimization. He has received six best paper awards at leading conferences in natural computation, GECCO and PPSN.

Xin Yao is a chair (professor) of computer science and the director of the Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), University of Birmingham, United Kingdom. His work received the 2001 IEEE Donald G. Fink Prize Paper Award, 2010 IEEE Transactions on Evolutionary Computation Outstanding Paper Award, 2010 BT Gordon Radley Award for Best Author of Innovation (Finalist), 2011 *IEEE Transactions on Neural Networks* Outstanding Paper Award, and many other best paper awards at conferences. He received the prestigious Royal Society Wolfson Research Merit Award in 2012 and the 2013 IEEE CIS Evolutionary Computation Pioneer Award. He was the editor-in-chief (2003-2008) of *IEEE Transactions on Evolutionary Computation*. He has been invited to give more than 70 keynote/plenary speeches at international conferences. His major research interests include search-based software engineering, evolutionary computation, and ensemble learning. He has more than 400 refereed publications in international journals and conferences. He is a fellow of the IEEE and a distinguished lecturer of IEEE Computational Intelligence Society (CIS).